2

THE PASM PARALLEL PROCESSING SYSTEM: HARDWARE DESIGN AND INTELLIGENT OPERATING SYSTEM CONCEPTS

A Thesis ~~Proposal~~

Submitted to the Faculty

of

Purdue University

by

Thomas Schwederski

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

July 1986

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

ABSTRACT

Many of today's scientific and industrial problems require enormous computing power. Since circuit switching speeds are reaching fundamental limits, avenues to speed up computations other than that using faster components are being explored. One such avenue is the use of parallelism. PASM is a dynamically reconfigurable SIMD/MIMD parallel processing system with up to 1,024 processing elements (PEs). It can be dynamically reconfigured to work as one or more SIMD (single instruction stream - multiple data stream) and/or MIMD (multiple instruction stream - multiple data stream) machines. A prototype with 30 MC68010 microprocessors, including 16 PEs in the computational engine, is being designed and constructed. The design of the prototype hardware is described, as well as the design tradeoffs that were made. Extending the current prototype by the addition of a Network Interface Unit (NIU) to each PE is proposed. Such an NIU significantly enhances interprocessor communication by offloading communication overhead from the PE's main CPU. One way to extend the prototype design to a system with 1,024 processors in the computational engine is presented.

The powerful reconfiguration capabilities of PASM can be fully utilized only if all tradeoffs influencing reconfiguration are known. Research is proposed that investigates which attributes of the PASM architecture, operating system software, and potential application programs affect both the cost and advantages of system reconfiguration. This knowledge can be incorporated in a knowledge base for an Intelligent Operating System that automatically configures and reconfigures the PASM system to achieve optimum performance.

THESIS MATERIAL PREVIOUSLY PUBLISHED BY THE AUTHOR

The author has previously published or submitted for publication portions of the work presented in this thesis proposal in the open literature. These publications are listed below.

[1] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Computer Society Spring Compcon 85,* February 1985, pp. 429-434.

[2] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems,* December 1985, pp. 603-612.

[3] T. Schwederski and H. J. Siegel, "Adaptable software for supercomputers," *Computer,* Vol. 19, February 1986, pp. 40-48.

[4] T. Schwederski, H. J. Siegel, E. J. Delp, A. Whinston, and L. H. Jamieson, "Modeling the PASM parallel processing system," *Proceedings of the SIAM 1986 National Meeting,* abstract, to appear, July 1986.

[5] T. Schwederski, D. G. Meyer, and H. J. Siegel, "Parallel Processing," in *Computer Architecture: Concepts and Systems,* V. Milutinovic, ed., Elvesier Science Publishing Co., New York, NY, to appear, 1986.

[6] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Tutorial on Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., to appear, 1986.

CHAPTER 1

INTRODUCTION

Mɛny of today's scientific and industrial problems require enormous computing power. Since circuit switching speeds are reaching fundamental limits, avenues to speed up computations other than that using faster components are being explored. One su:h avenue is the use of parallelism. PASM is a dynamically reconfigurable SIMD/MIMD parallel processing system with up to 1,024 processing elements (PEs). It can be dynamically reconfigured to work as one or more SIMD (single instruction stream - multiple data stream) and/or MIMD (multiple instruction stream - multiple data stream) machines. A prototype with 30 MC68010 microprocessors, including 16 PEs in the computational engine, is being designed and constructed.

Various aspects of the PASM system and its prototype are topics of this proposal. In Chapter 2, literature related to the proposed research is surveyed. The first part of the chapter gives an overview of parallel processing issues: concepts, software, and systems. The second part introduces adaptable software for supercomputers. Adaptable software is essential to fully utilize reconfigurable supersystems. In Chapter 3 the PASM system and its prototype are overviewed. The hardware design of the prototype and the design tradeoffs that were made are described in Chapter 4. At the time of this writing, all hardware has been designed and debugged, and the prototype is expected to become operational in July 1986. In Chapter 5, an extension of the current prototype by the addition of a Network Interface Unit (NIU) to each PE is proposed. The overhead of sending messages through the Inter-PE Network is analyzed, and it is shown that an NIU significantly enhances inter-PE communication by offloading communication overhead from the PE's main CPU. An approach for implementing a PASM system with 1,024 processors in the computational engine is shown in Chapter 6.

The powerful reconfiguration capabilities of PASM can be fully utilized only if all tradeoffs influencing reconfiguration are known. In Chapter 7, research is proposed that investigates which attributes of the PASM architecture, operating system software, and potential application programs affect both the cost and advantages of system reconfiguration. This information can be incorporated in a knowledge base for an Intelligent Operating System that automatically configures and reconfigures the PASM system to achieve optimum performance.

CHAPTER 2

SURVEY OF RELATED LITERATURE

A survey of parallel processing concepts, software, and systems is in Appendix A.1. A survey of adaptable software that is essential to fully utilize reconfigurable supersystems is in Appendix A.2.

CHAPTER 3

AN OVERVIEW OF THE PASM PARALLEL PROCESSING SYSTEM

3.1 Introduction

This is an overview of the design for the thousand-processor PASM system and the 30-processor prototype being built to validate the system design concepts. Parallelism is one way to increase computational speed. Different modes of parallelism can be employed in a computer system. The *SIMD (single instruction stream - multiple data stream)* mode [Fly66] typically uses a set of N processors, N memories, an interconnection network, and a control unit (e.g., Illiac IV [BoD72], STARAN [Bat76, Bat77], CLIP4 [Fou81], MPP [Bat82]). The control unit broadcasts instructions to the processors and all active (enabled) processors execute the same instruction at the same time. Each processor executes instructions using data taken from a memory with which only it is associated. The interconnection network allows interprocessor communication. An *MSIMD (multiple-SIMD) system* is a parallel processing system which can be structured as one or more independent SIMD machines of various sizes (e.g., MAP [Nut77]). The Illiac IV was originally designed as an MSIMD system [BaB68]. The *MIMD (multiple instruction stream - multiple data stream)* mode [Fly66] typically consists of N processors and N memories, where each processor can follow an independent instruction stream (e.g., Ultracomputer

This chapter was co-authored by H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV [SiS86].

[GoG83], BBN Butterfly [CrG85], Cosmic Cube [Sei85], RP3 [PfB85]). As with SIMD architectures, there is a multiple data stream and an interconnection network. A *partitionable SIMD/MIMD system* is a parallel processing system which can be structured as one or more independent SIMD and/or MIMD machines of various sizes (e.g., TRAC [SeU80]).

PASM is a partitionable SIMD/MIMD machine being designed at Purdue University to be a large-scale dynamically reconfigurable multimicroprocessor system [SiS81]. It is a special-purpose system aimed at exploiting the parallelism of image understanding tasks. PASM is being developed using a variety of problems in image processing and pattern recognition to guide the design choices. It can also be applied to related areas such as speech understanding and biomedical signal processing.

PASM is to serve as a research tool for experimenting with parallel processing. The design attempts to incorporate the needed flexibility for studying large-scale SIMD and MIMD parallelism, while keeping system costs "reasonable." Portions of PASM have been simulated and a prototype is under construction.

In Section 3.2, the PASM architecture is overviewed. Section 3.3 gives some examples of how PASM can be used for image processing. The PASM prototype is described in Section 3.4. A summary of the parallel programming language and distributed operating system development for PASM is given in Section 3.5. The advantages of some of the features of PASM are discussed in Section 3.6.

3.2 The PASM Architecture

3.2.1 The Parallel Computation Unit

A block diagram of the basic components of PASM is shown in Figure 3-1. The *Parallel Computation Unit* (Figure 3-2) contains $N=2^n$ processors, N memory modules, and an interconnection network. The ***processors*** are microprocessors that perform the actual SIMD and MIMD computations. The ***memory modules*** are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. Each PE can operate in both the SIMD and MIMD modes of parallelism. A memory module is connected to each processor to form a processor - memory pair called a *Processing Element (PE)*. The N PEs are numbered from 0 to N−1 and each PE knows its number (address). A pair of memory units is used for each memory module to allow data to be moved between one memory unit and secondary storage (the Memory Storage System) while the processor operates on data in the other memory unit. The PASM N=16 prototype under construction uses Motorola MC68010 processors; the final N=1024 system, which the architecture is designed for, may employ custom VLSI processors specially designed for parallel image understanding. The ***interconnection network*** provides a means of communication among the PEs. Two types of multistage interconnection networks are being considered for PASM: the Generalized Cube and the Augmented Data Manipulator (ADM) [Sie85, SiM81b, SiM81a]. The ADM network is more flexible but is more complex. Features of the Generalized Cube network will be described to familiarize the readers with the properties of multistage networks.

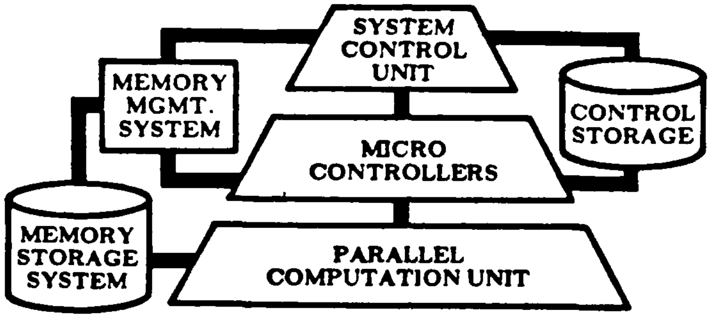


Figure 3-1. Block diagram overview of PASM.

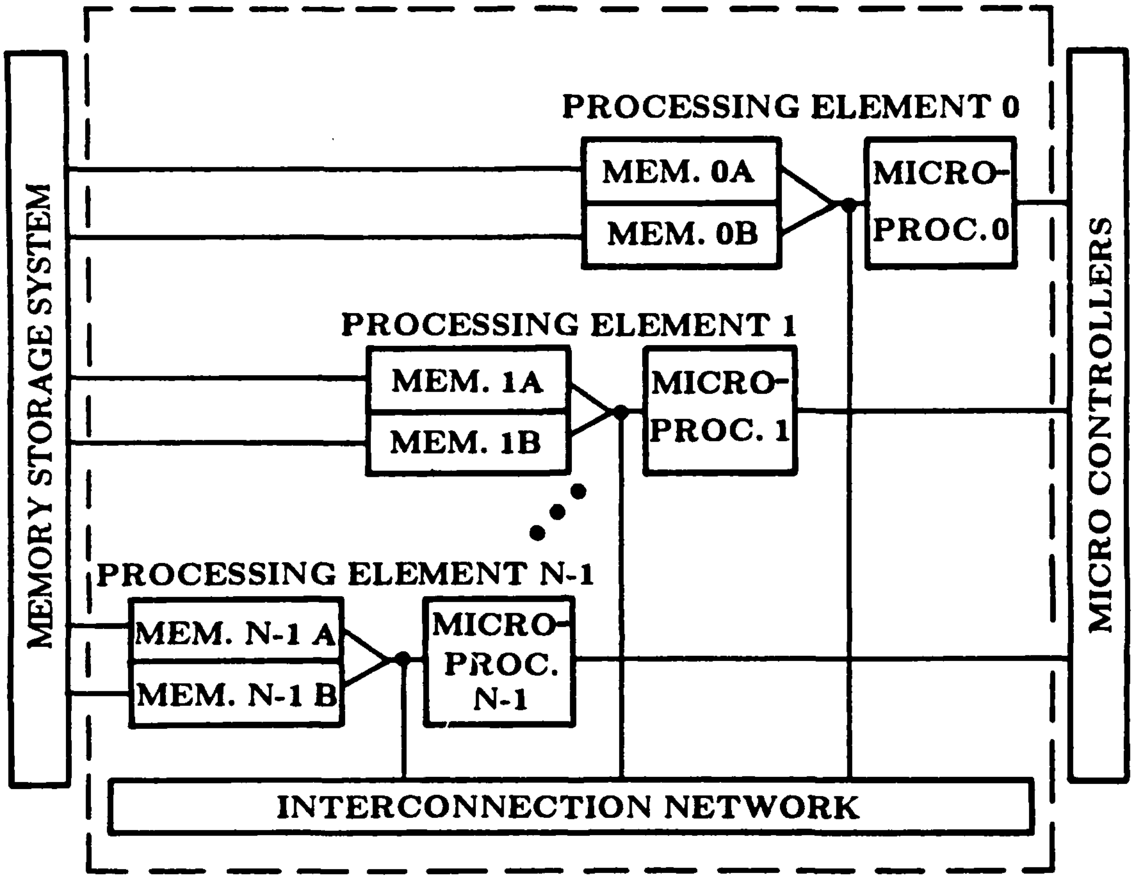


Figure 3-2. Parallel Computation Unit.

The *Generalized Cube* network is representative of the multistage cube-type class of networks which include the baseline [WuF80], delta [Pat81], indirect binary n-cube [Pea77], omega [Law75], and SW-banyan (S=F=2) [GoL73]. This class has been used or proposed for use in systems such as STARAN [Bat76, Bat77], Ultracomputer [GoG83], BBN Butterfly [CrG85], and RP3 [PfB85]. The Cube has N inputs and N outputs. It is shown in Figure 3-3a for N=8. PE i, $0 \leq i < N$, would be connected to input port i and output port i of the unidirectional network. The Cube topology has $n = \log_2 N$ stages, where each stage consists of a set of N lines connected to N/2 interchange boxes. Each *interchange box* is a two-input, two-output device. The labels of the input/output *(I/O)* lines entering the upper and lower inputs of an interchange box are used as the labels for the upper and lower outputs, respectively. Each interchange box can be set individually to one of the four legitimate states shown in Figure 3-3a. Figure 3-3b, c, and d illustrate one-to-one, broadcast, and permutation connections, respectively. Note that many one-to-one and/or broadcasts can occur simultaneously.

The connections in this network are based on the cube interconnection functions [Sie77, Sie79]. Let $P = p_{n-1} \cdots p_1 p_0$ be the binary representation of an arbitrary I/O line label. Then the n cube interconnection functions can be defined as:

$$\mathrm{cube}_i(p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_1 p_0$$

where $0 \leq i < n$, $0 \leq P < N$, and $\bar{p}_i$ denotes the complement of p_i. This means that the cube_i interconnection function connects P to $\mathrm{cube}_i(P)$, where $\mathrm{cube}_i(P)$ is the I/O line whose label differs from P in just the i-th bit position. Stage i of the Cube topology contains the cube_i interconnection function, i.e., it pairs I/O lines that differ only in the i-th bit position.

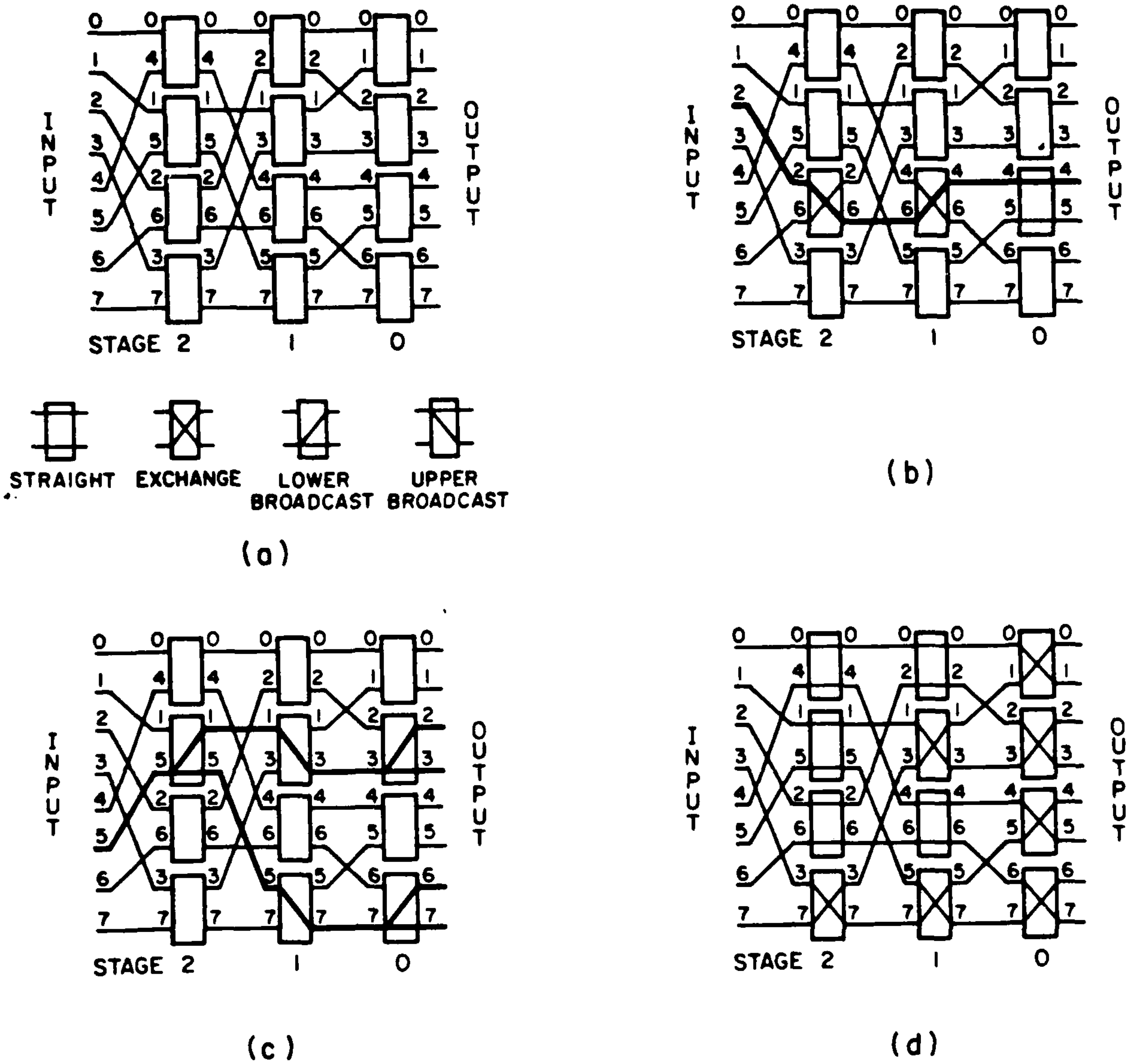


Figure 3-3. (a) Generalized Cube topology, shown for N=8. (b) Example one-to-one connection (input 2 to output 4). (c) Example broadcast connection (input 5 to outputs 2, 3, 6, and 7). (d) Example permutation connection (input i to output i+1 mod N). [Sie85]

Routing tags are used as headers on messages; they (1) control each interchange box individually, and (2) allow network control to be distributed among the PEs. The n-bit routing tag for one-to-one connections is computed from the input port number and desired output port number. Let S be the source address (input port number) and D be the destination address (output port number). Then the routing tag $T = S \oplus D$ (where "$\oplus$" means bitwise "exclusive-or"). Let $t_{n-1} \ldots t_1 t_0$ be the binary representation of T. An interchange box in the network at stage i need only examine t_i. If $t_i=1$, an exchange is performed, and if $t_i=0$, the straight connection is used. For example, if N=8, S=010, and D=100, then T=110. The corresponding stage settings are exchange, exchange, straight (see Figure 3-3b). Because the exclusive-or operation is commutative, the incoming routing tag is the same as the return tag. Since the destination PE has the routing tag to the source PE, it is easy to perform handshaking if desired. The address of the source PE can be computed by the destination PE using $S = D \oplus T$. Routing tags that can be used for broadcasting data are an extension of this scheme [SiM81b].

The *partitionability* of a network is its ability to divide the system into independent subsystems of different sizes. The Cube network can be partitioned into independent subnetworks of various sizes, where each subnetwork of size $N' \leq N$ has all of the connection properties of a Cube network built to be of size N'. In PASM, the partitioning is accomplished by requiring that the addresses of all of the I/O ports in a partition of size 2^i agree (have the same values) in their low-order $n-i$ bit positions. For example, in Figure 3-4 subnetwork A consists of ports 0, 2, 4, and 6, and subnetwork B consists of ports 1, 3, 5, and 7. All ports in subnetwork A have a 0 in the low-order bit position; all ports in subnetwork B have a 1 in the low-order bit position. By setting all of

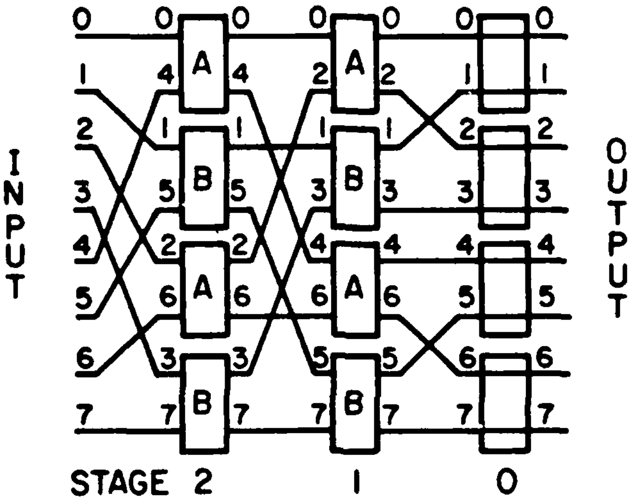


Figure 3-4. Cube network of size eight partitioned into two subnetworks of size four based on the low-order bit position. [Sie85]

the interchange boxes in stage 0 to straight, the two subnetworks are isolated. This is because stage 0 is the only stage which allows input ports which differ in their low-order bit to exchange data. Each subnetwork can be separately further subdivided, resulting in subnetworks of various sizes. The routing tag scheme can be used to enforce the partitioning by logically AND-ing the tags with masks to force to 0 tag positions which correspond to stages whose interchange boxes should be forced to the straight state. The network partitioning property allows the set of PASM PEs to be divided into independent partitions of various sizes.

The Cube network as presented here has the disadvantage that only a single path exists from any source to any destination. Thus, a single fault in the network makes many source-destination paths unavailable. The Extra Stage Cube network [AdS82], a single-fault tolerant variation of the Cube, overcomes this problem by introducing an additional "extra" stage. Figure 3-5a shows the expanded network. The extra (input) stage performs cube_2. This is followed by the "normal" Cube network stages which perform cube_0, cube_1, and cube_2. The input (extra) and output stages can be bypassed as shown in Figure 3-5b through 3-5g. Under normal fault-free operation, the extra stage is disabled, i.e., the bypass circuitry is used. If a fault is present in the extra stage, the bypass circuitry is employed as in the no-fault case and network operation is not affected (see Figure 3-6a). If a fault occurs in an intermediate stage, both the extra stage and the output stage are enabled. This establishes two paths for every source-destination pair, and these paths are distinct except for the extra stage and output stage boxes. This is demonstrated in Figure 3-6b. Since there are two distinct paths, one must be fault free when a single fault occurs in an intermediate stage. If a fault occurs in the output stage, its

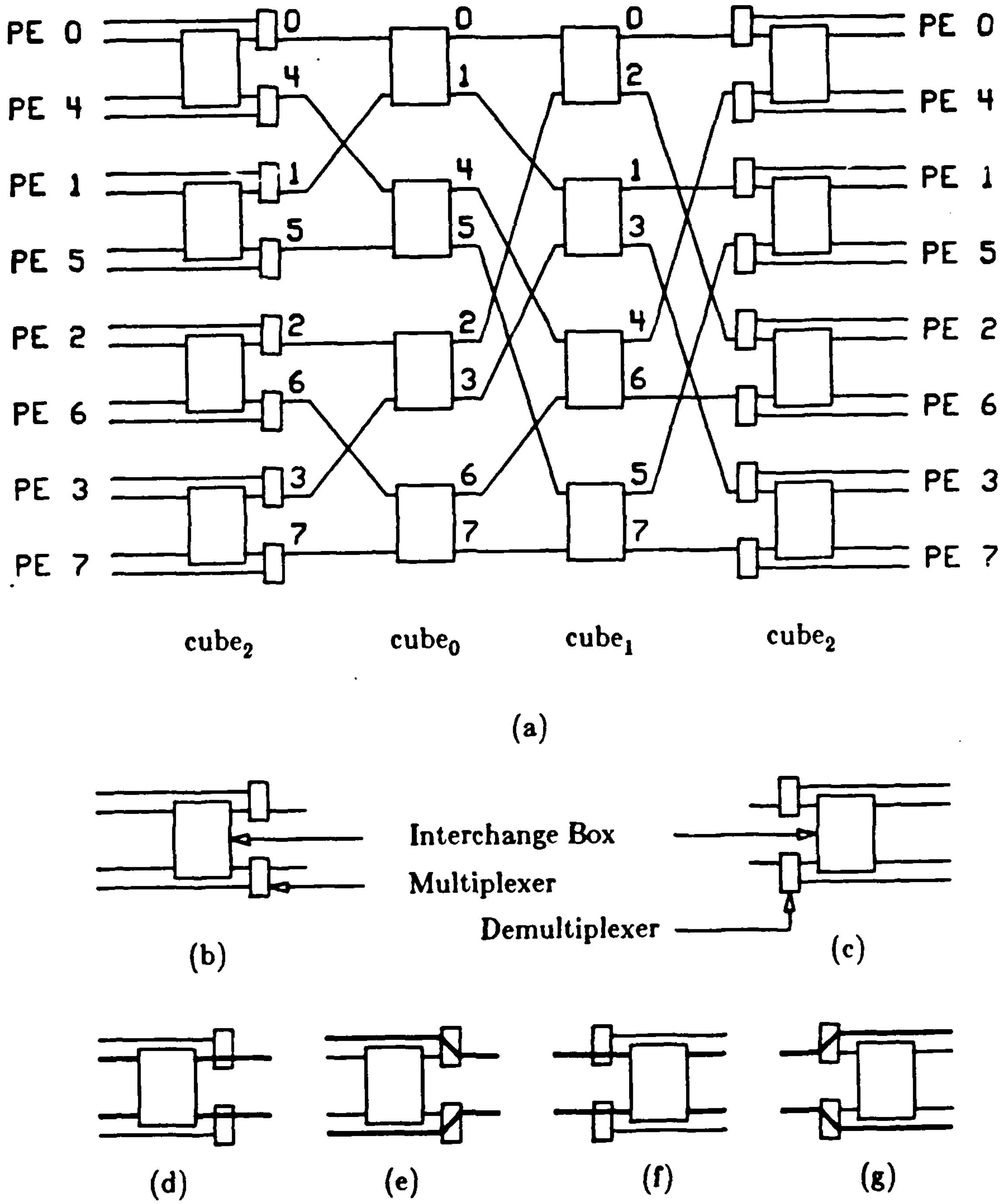
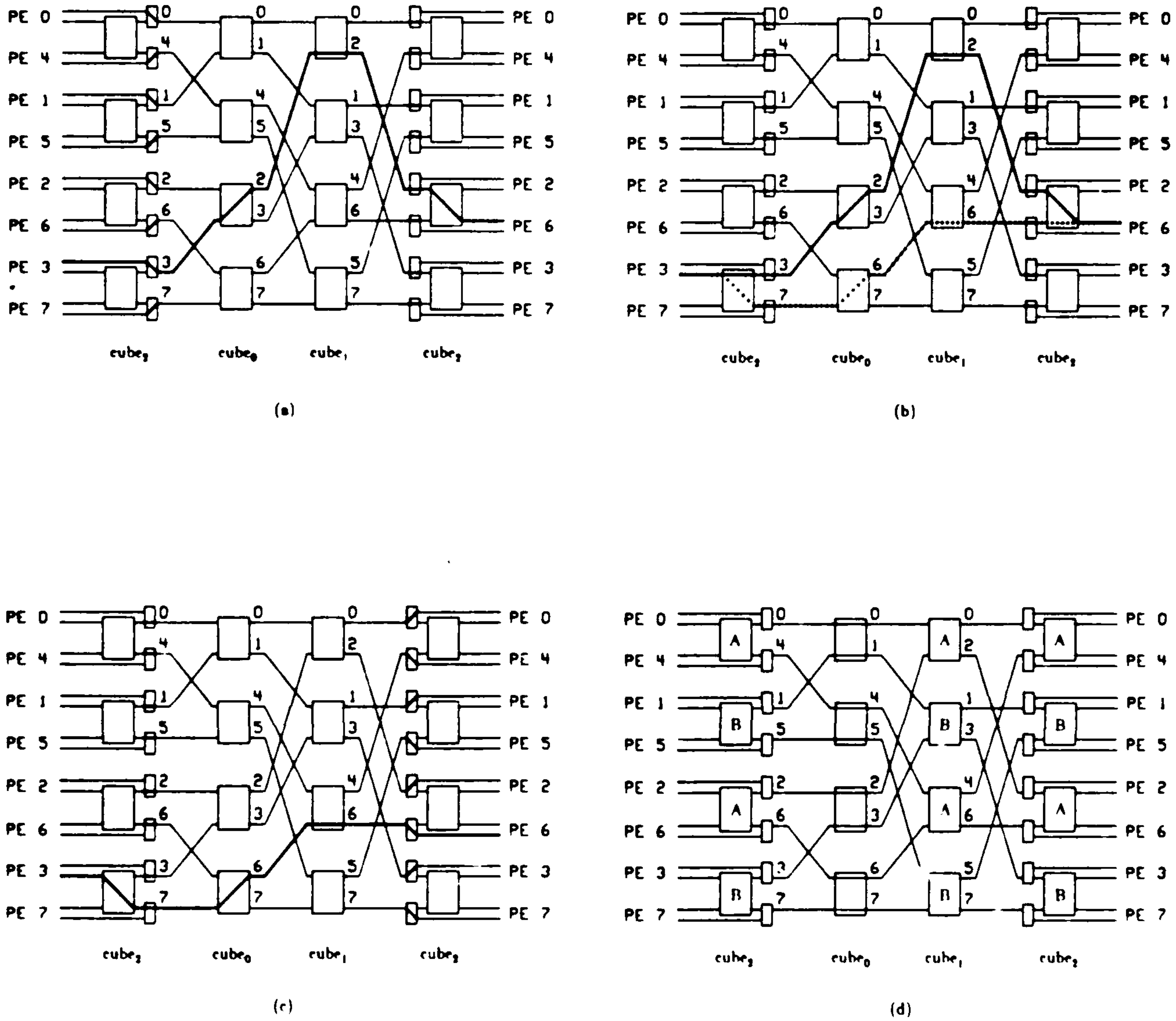


Figure 3-5 (a) The Extra Stage Cube Network topology, shown for N=8. (b) Detail of input interchange box. (c) Detail of output interchange box. (d) Input stage enabled. (e) Input stage disabled. (f) Output stage enabled. (g) Output stage disabled.



(a)

(b)

(c)

(d)

Figure 3-6.

Routing in the Extra Stage Cube Network for N=8. (a) Data transfer from PE 3 to PE 6, extra stage disabled (no fault or fault in the extra stage). (b) Data transfer from PE 3 to PE 6, extra stage and output stage enabled (fault in an intermediate stage). (c) Data transfer from PE 3 to PE 6, extra stage enabled and output stage disabled (fault in the output stage). (d) Partitioning the network into even ("A") and odd ("B") numbered PEs.

function (i.e., $cube_2$) is performed by the extra stage and the bypass circuitry in the output stage is used to bypass the fault there (see Figure 3-6c). The Extra Stage Cube network provides the same partitioning capabilities as does the regular Cube network, even in the presence of a fault. A partitioning example is shown in Figure 3-6d. In addition to being single-fault tolerant, it has been shown to be very robust under multiple faults [AdS84].

3.2.2 The Micro Controllers

The *Micro Controllers (MCs)* (Figure 3-7) are a set of microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. There are $Q=2^q$ MCs, physically addressed (numbered) from 0 to Q–1. Each MC controls N/Q PEs PASM is being designed for Q=32 (Q=4 in the prototype). The MCs are the multiple control units needed in order to have a partitionable SIMD/MIMD system. Each MC memory module consists of a pair of memory units so that memory loading and computations can be overlapped. In SIMD mode, each MC fetches instructions and common data from its memory module, executing the control flow instructions (e.g., branches) and broadcasting the data processing instructions to its PEs. In MIMD mode, each MC gets from its memory instructions and common data for coordinating its PEs.

The reconfiguration rule for PASM is that all PEs in a partition of size 2^p must agree in their low order p address bits. Thus the physical addresses of the N/Q processors which are connected to an MC must all have the same low-order q bits so that they can form a partition. The value of these low-order q bits is the physical address of the MC. An SIMD machine partition of

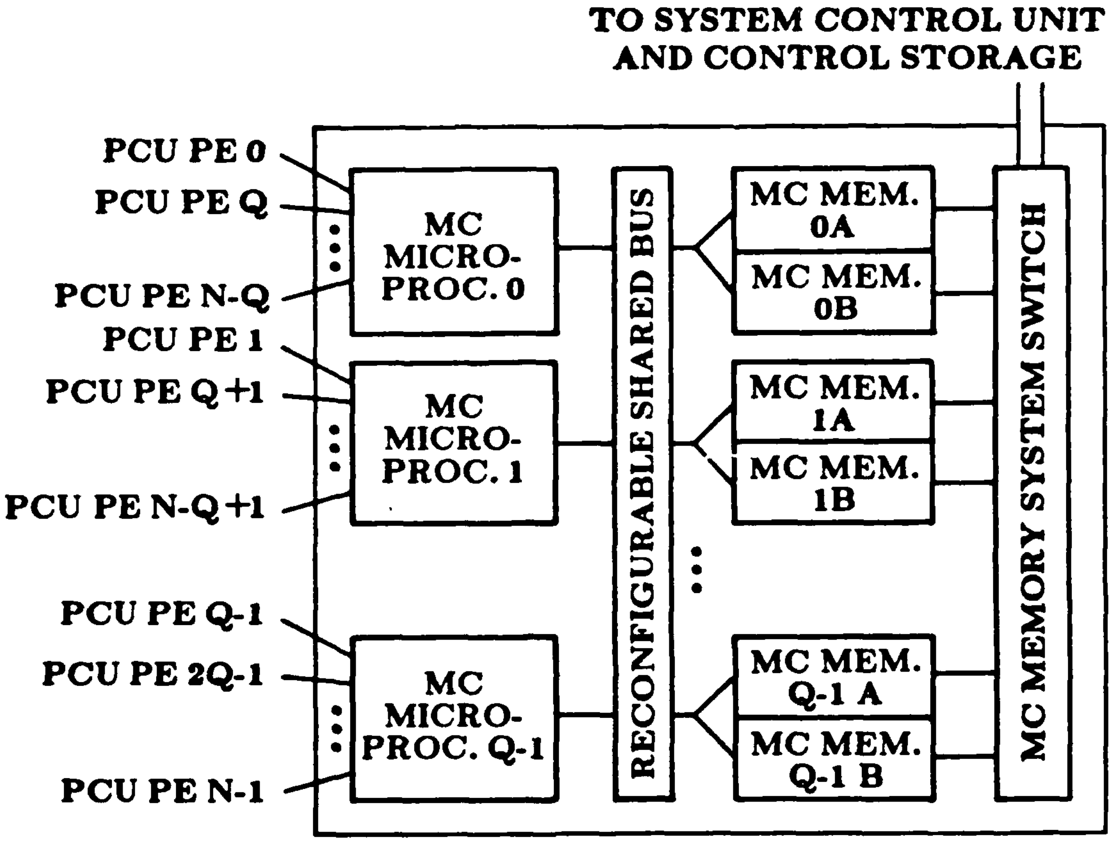


Figure 3-7. PASM Micro Controllers (MCs). "PCU" is Parallel Computation Unit.

size RN/Q, where $R=2^r$ and $0 \leq r \leq q$, is obtained by having R MCs use the same instructions and synchronizing the MCs. One way to provide the R MCs with the same instructions is by loading each of the memory modules associated with the R MCs with the same program (if this scheme is used, the reconfigurable bus in Figure 3-7 is not needed and MC microprocessor i is directly connected to MC memory module i). The physical addresses of these MCs must have the same low-order q−r bits so that all of the PEs in the partition have the same low-order q−r physical address bits. Similarly, an MIMD machine partition of size RN/Q is obtained by combining the efforts of the PEs associated with R MCs which have the same low-order q−r physical address bits. In MIMD mode, the MCs are used to help coordinate the activities of their PEs. Q is the maximum number of partitions allowable, and N/Q is the size of the smallest partition.

As an alternative to loading the same program in R MC memories for an SIMD machine partition, the MC processors and MC memories may be connected by a shared reconfigurable ("shortable") bus [ArP76, KaK79], as shown in Figure 3-8. The MCs must be ordered on the bus in terms of the bit reverse of their addresses due to the partitioning rules. The advantages of the MC connection scheme are that it provides more program space for jobs using multiple MCs and provides a degree of fault tolerance, since known-faulty MC memory modules could be ignored. This scheme has the disadvantages of adding propagation delay through the bus switches and adding hardware complexity would could reduce reliability. These tradeoffs are currently being studied.

The PEs within each partition are assigned *logical addresses*. Given a machine partition of size RN/Q, the processors and memory modules for this

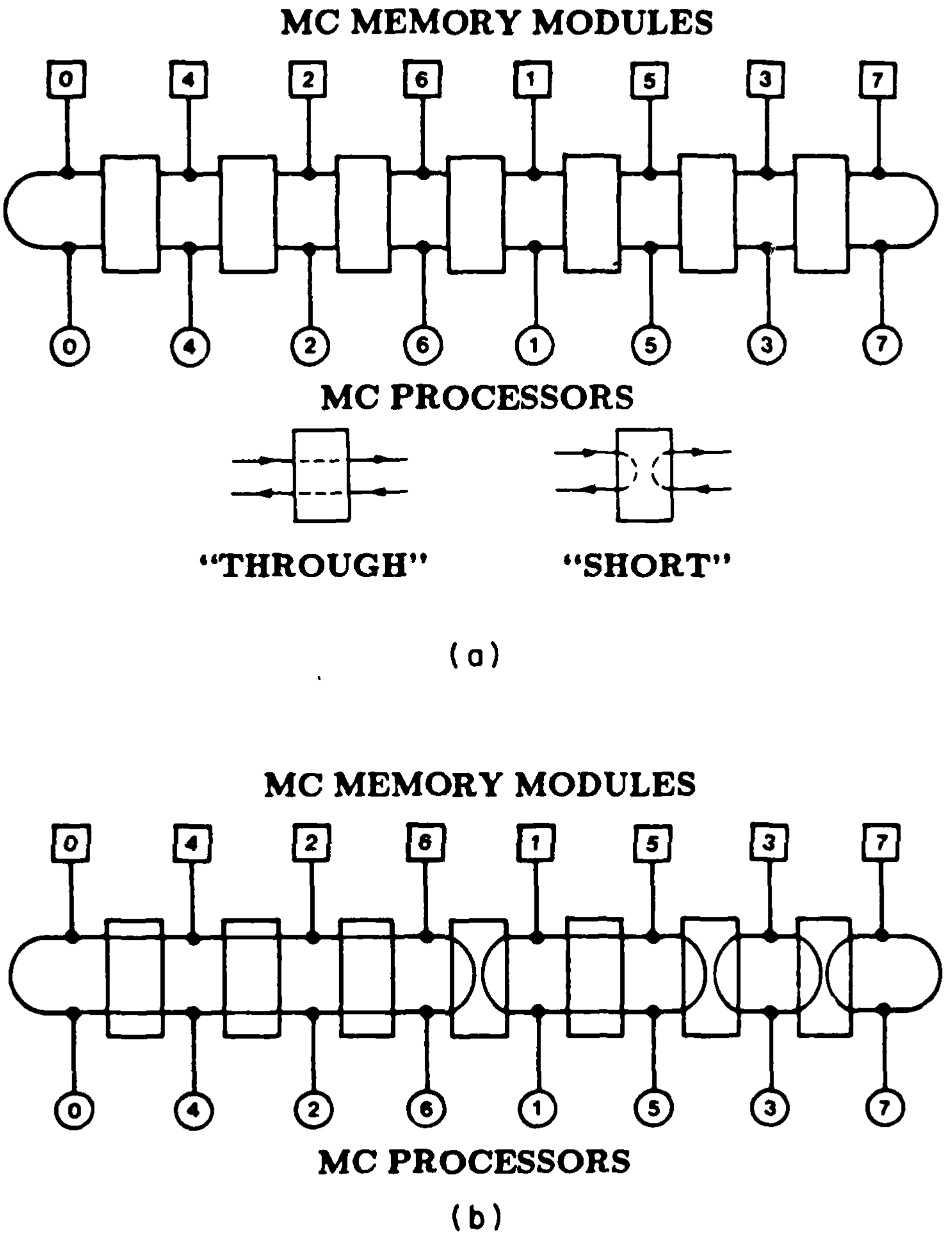


Figure 3-8. (a) Reconfigurable shared bus scheme for interconnecting MC processors and MC memory modules, shown for Q=8, where each box can be set to "through" or "short." (b) Bus set for MCs 0, 2, 4, and 6 forming one machine partition, MCs 1 and 5 forming a second partition, MC 3 forming a third partition, and MC 7 forming a fourth partition.

partition have logical addresses (numbers) 0 to (RN/Q)−1, $R=2^r$, $0 \leq r \leq q$. The logical number of a PE is the high-order r+n−q bits of its physical number. Similarly, the MCs assigned to the partition are logically numbered (addressed) from 0 to R−1. For R>1, the logical number of an MC is the high-order r bits of its physical number. The PASM operating system chooses the partition to be used for executing a user's task. A system user only deals with logical addresses which are translated to physicai addresses by the operating system.

A *masking scheme* is used in SIMD mode for determining which PEs will be active, i.e., execute instructions broadcast to them by their MC. PASM uses PE address masks and data conditional masks.

The *PE address masking* scheme uses an n-position mask to specify which of the N PEs are to be activated. Each position of the mask contains either a 0, 1, or X ("don't care") and the only PEs that are enabled are those whose address matches the mask: *0* matches *0*, *1* matches *1*, and *either 0 or 1* matches X. Square brackets denote a mask. Superscripts are used as repetition factors. For example, "MASK $[X^{n-1}0]$" activates all even-numbered PEs and "MASK $[0^{n-i}X^i]$" activates PEs 0 to 2^i−1. A *negative PE address mask* is similar to a regular PE adaress mask, except that it activates all those PEs which do not match the mask. Negative PE address masks are prefixed with a minus sign to distinguisk them from regular PE address masks. For example, for N=8, "MASK [−0X1]" activates all PEs except 1 and 3. PE address masks are specified in the SIMD program.

Data conditional masks are the implicit result of performing a conditional branch dependent on local data in an SIMD machine environment, where the result of different PEs' evaluations may differ. They are used when the decision to enable and disable PEs is made at execution time. As a result of a conditional *where statement* oı the form

where <data–condition> *do* · · · *elsewhere* · · ·

each PE sets a flag to activate itself for either the "do" or the "elsewhere," but not both. The execution of the "elsewhere" statements must follow the "do" statements; i.e., the "do" and "elsewhere" statements cannot be executed simultaneously. For example, as a result of executing the statement:

where A < B *do* C ← A *elsewhere* C ← B

each PE loads its C register with the minimum of its A and B registers, i.e., some PEs execute "C ← A," and then the rest execute "C ← B." This type of masking is used in such machines as the Illiac IV [BaB68] and PEPE [CrG72]. "Where" statements can be nested using a run-time control stack.

There are instructions which examine the collective status of all of the PEs in an SIMD machine partition, such as "if any," "if all," and "if none." These instructions change the flow of control of the program at execution time depending on whether any or all processors in the SIMD machine partition satisfy some condition. For example, if each PE is processing data from a different section of an image, but all PEs are looking for enemy tanks, it is desirable to know "if any" of the PEs have discovered a tank. This requires communication among the MCs comprising the SIMD machine partition. There is a set of buses shared by MCs for this purpose.

3.2.3 Control Storage

Control Storage contains the programs for the MCs. It consists of a secondary storage device and a microprocessor for managing the file system on the device. The Control Storage processor acts as a file server by responding to requests to load programs into the MC memory units.

3.2.4 The Memory Storage System

The *Memory Storage System* provides secondary storage space for the Parallel Computation Unit for the data files in SIMD mode, and for the data and program files in MIMD mode. It consists of N/Q independent *Memory Storage Units*, numbered from 0 to (N/Q)−1. Each Memory Storage Unit is connected to Q PE memory modules. For $0 \leq i < N/Q$, Memory Storage Unit i is connected to those PE memory modules whose physical addresses have the value i in their n−q high-order bits. Recall that, for $0 \leq k < Q$, MC k is connected to those PEs whose physical addresses have the value k in their q low-order bits. This is shown for N=32 and Q=4 in Figure 3-9. Like Control Storage, each Memory Storage Unit has a microprocessor to manage the file system on the physical disk device and to service file system requests by loading/unloading PE memory units.

A machine partition of RN/Q PEs, $R = 2^r$, $0 \leq r \leq q$, logically numbered from 0 to (RN/Q)−1, requires only R parallel block loads if the data for the PE memory module whose high-order n−q logical address bits equal i is loaded into Memory Storage Unit i. This is true no matter which group of R MCs (which agree in their low-order q−r physical address bits) is chosen. As an example, consider Figure 3-9, and assume a partition of size 16 is desired. The data for the PE memory modules whose logical addresses are 0 and 1 is loaded into Memory Storage Unit 0, for memory modules 2 and 3 into Unit 1, etc. Assume the partition of size 16 is chosen to consist of the PEs connected to MCs 1 and 3. Given this assignment of MCs, the PE memory module whose physical

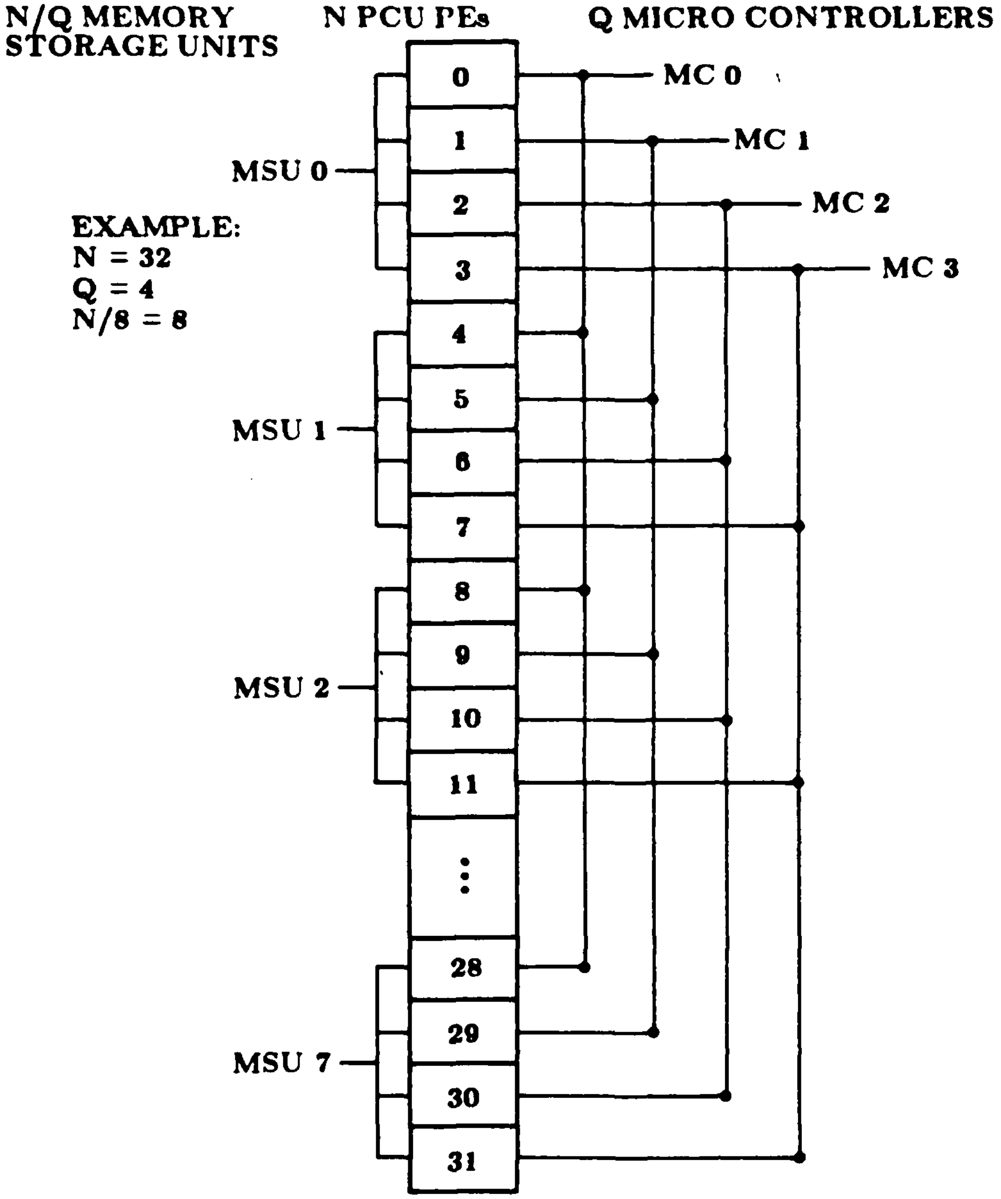


Figure 3-9. Organization of the Memory Storage System, shown for N=32 and Q=4. "MSU" is Memory Storage Unit. "PCU" is Parallel Computation Unit.

address is $2i+1$ has logical address i, $0 \leq i < 16$. The Memory Storage Units first simultaneously load PE memory modules physically addressed 1, 5, 9, 13, 17, 21, 25, and 29 (logically addressed 0, 2, 4, 6, 8, 10, 12, and 14), and then simultaneously load PE memory modules physically addressed 3, 7, 11, 15, 19, 23, 27, and 31 (logically addressed 1, 3, 5, 7, 9, 11, 13, and 15). No matter which pair of MCs is chosen (i.e., MCs 1 and 3, or MCs 0 and 2), only two parallel block loads are needed. This same approach can be taken if only $(N/Q)/2^d$ distinct Memory Storage Units are available, where $0 \leq d \leq n-q$. In this case, however, $R2^d$ parallel block loads are required instead of just R to load a machine partition of RN/Q PEs.

3.2.5 The Memory Management System

The *Memory Management System* controls the transferring of files between the Memory Storage System and the PEs. It is composed of a separate set of microprocessors dedicated to performing tasks in a distributed fashion. This distributed processing approach is chosen in order to provide the Memory Management System with a large amount of processing power at low cost. The division of tasks chosen is based on the main functions which the Memory Management System must perform, including: (1) fielding file system requests from the System Control Unit, MCs, or PEs (via their MCs) and passing them on to the appropriate Memory Storage Units; (2) scheduling Memory Storage System data transfers; (3) controlling input/output operations involving peripheral devices and the Memory Storage System; (4) enforcing a consistent file naming and placement policy for files distributed over the Memory Storage Unit disks; and (5) controlling the Memory Storage System bus.

3.2.6 The System Control Unit

The *System Control Unit* is responsible for the overall coordination of the activities of the other components of PASM. The types of tasks the System Control Unit performs include program development, job scheduling (choosing a machine partition for a user job), and coordination of the loading of the PE memory modules from the Memory Storage System with the loading of the MC memory modules from Control Storage. By carefully choosing which tasks should be assigned to the System Control Unit and which should be assigned to other system components (e.g., the MCs and Memory Management System), the System Control Unit can work effectively and not become a bottleneck. For the N=1024 PASM, the System Control Unit may consist of several processors in order to perform all of its functions efficiently. In the N=16 prototype, the System Control Unit is a microprocessor and the program development functions are performed by the host computer network.

3.3 Parallel Image Processing Algorithms

3.3.1 Introduction

A number of SIMD and MIMD algorithms to perform common image processing tasks have been developed by our group. Image processing algorithms which have been structured for parallel execution include: image smoothing, histogramming, 2-D FFT calculation, local area histogram equalization, local area brightness and gain control, feature extraction, maximum likelihood classification, contextual statistical classification, image correlation (convolution, filtering), block truncation coding, resampling, rectification, rotation, translation, scaling, elevation/location determination, median filtering, Sobel edge sharpening, clustering feature enhancement, scene segmentation, Karhunen-Loeve transformation, 3-D shape analysis using Fourier descriptors, raster-to-vector and vector-to-raster conversion (image thinning, vectorization), and a computer vision task including edge labeling, perimeter, area, center of mass, and hole count computations. These have been analyzed in terms of machine-size/problem-size relationships, processor capability needed for efficient execution, memory requirements, and inter-PE communication requirements. Different algorithm strategies have been explored and compared. From these studies, we are assessing the ways in which parallelism can be used for vision.

In this section three brief examples of parallel image processing algorithms are given. The first is a simple SIMD image smoothing algorithm, the second a more complex SIMD histogramming algorithm, and the third an SIMD/MIMD contour extraction algorithm.

3.3.2 Image Smoothing

As an example of a simple parallel algorithm, consider the *smoothing of an image* [SiS81]. The algorithm described here smooths a gray level input image. The algorithm has "I" as an input image and "S" as an output image. Assume both I and S contain 512-by-512 *pixels* (picture elements), for a total of 512^2 pixels each. Each point of I is an eight-bit unsigned integer representing one of 256 possible gray levels. The gray level of each pixel indicates how "dark" that pixel is, where 0 means white and 255 means black. Each point in the smoothed image, S(i,j), is the average of the gray levels of I(i,j) and its eight nearest neighbors, I(i−1,j−1), I(i−1,j), I(i−1,j+1), I(i,j−1), I(i,j+1), I(i+1,j−1), I(i+1,j), and I(i+1,j+1). The top, bottom, left, and right edge pixels of S are not calculated since their corresponding pixels in I do not have eight adjacent neighbors.

Consider how this could be implemented on an SIMD machine with N = 1024 PEs, logically arranged as an array of 32-by-32 PEs as shown in Figure 3-10. Each PE stores a 16-by-16 subimage block of the 512-by-512 image I. Specifically, PE 0 stores the pixels in columns 0 to 15 of rows 0 to 15, PE 1 stores the pixels in columns 16 to 31 of rows 0 to 15, and so on. Each PE smooths its own subimage, with all PEs doing this simultaneously. At the edges of each 16-by-16 subimage, data must be transmitted between PEs in order to calculate the smoothed value. The necessary data transfers are shown for PE J in Figure 3-10. Transfers between different PEs can occur simultaneously; e.g., when PE J−1 sends its upper right corner pixel to PE J, PE J can send its upper right corner pixel to PE J+1, PE J+1 can send its upper right corner pixel to PE J+2, etc.

In order to perform a smoothing operation on a 512-by-512 image by the parallel smoothing of 1024 subimage blocks of size 16-by-16, 16^2 = 256 parallel smoothing operations are performed. As described above, the neighbors of the subimage edge pixels must be transferred in from adjacent PEs. Using either the Cube or ADM networks, the total number of parallel data element transfers needed is (4 * 16) + 4 = 68: 16 for each of the top, bottom, left side, and right side edges, and four for the corners (see Figure 3-10). The corresponding serial algorithm needs no data transfers between PEs, but 512^2 = 262,144 smoothing calculations must be performed. If no data transfers were needed, the parallel algorithm would be faster than the serial algorithm by a factor of 262,144/256

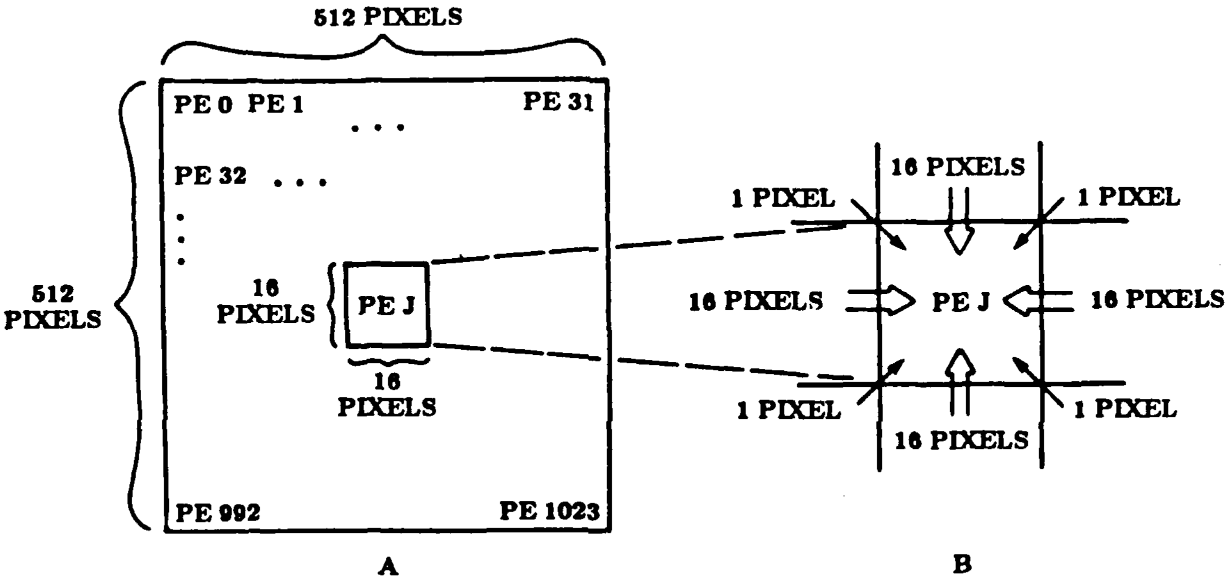


Figure 3-10. Data allocation and inter-PE transfers for the image smoothing algorithm.

= 1024 = N. If the inter-PE data transfer time is included and it is assumed that each parallel data transfer requires at most as much time as one smoothing operation, then the time factor improvement is 262,144/324 = 809. The inter-PE transfer time approximation is a conservative one. Thus, the overhead of the 68 inter-PE transfers that must be performed in the SIMD machine is negligible compared to the reduction from 262,144 to 256 smoothing operations.

3.3.3 Global Histogramming

As an example of a parallel algorithm that is not a straightforward decomposition of the corresponding serial algorithm (as the smoothing example was), consider an SIMD algorithm for computing the *global histogram of an image* [SiS81]. Assume there are B = 2^b = 256 bins in the histogram, N=1024, and the image is 512-by-512 pixels. The B-bin histogram of the image contains a j in bin i if exactly j of the pixels have a gray level of i, $0 \leq i < B$. Assume the image is equally distributed among the 1024 PEs, i.e., each PE has $512^2/1024$ pixels, and $B \leq 512^2/1024$. Since the image is distributed over 1024 PEs, each PE calculates a B-bin histogram based on its subimage. Then these "local" histograms are combined using the techniques described below.

The method of *recursive doubling* [Sto80] is used in the combining procedure. Consider the problem of summing up all elements of a vector. In a serial computer, a variable would be initialized to the value of the first element of the vector, and all subsequent elements would be added to this variable. If the vector contained N elements, N−1 additions are required. Assume that on a parallel computer the vector is distributed among processors, i.e., each of N processors holds one element. The recursive doubling procedure is illustrated in Figure 3-11. Assume that N is a power of 2; in the Figure, N=8. In the first step, all odd numbered PEs (i.e., 1, 3, 5, 7) send their vector element to an even numbered PE (i.e., 1 to 0, 3 to 2, etc.). All even numbered PEs then add the value they received to their own vector element, forming N/2 (=4) partial results. The odd numbered PEs do not participate an' are disabled. In the next step, this procedure is repeated. PE 2 forwards its partial result to PE 0, while PE 6 sends its result to PE 4. PEs 0 and 4 add the new value to their respective partial results, and all other PEs are disabled. In the last step, PE 4 sends its partial result to PE 0. PE 0 adds this value to its own partial result, and the sum is found. The overall procedure required three transfers and three additions. In general, for N a power of two, $\log_2 N$ additions and data transfer steps are required.

After the local histograms are calculated, n steps are used to combine them. In the first b steps, each block of B PEs performs B simultaneous recursive doublings to compute the histogram for the portion of the image contained in the block (see Figure 3-12). In the first step, each block of PEs is divided in half such that the PEs with the lower addresses form one group, and the PEs with the higher addresses form another. Each group accumulates the sums for half of the bins, and sends the bins it is not accumulating to the other group. The "lower-numbered group" accumulates the sums for the first half of the bins while the "higher-numbered group" accumulates the second half of the bins. For each successive merging step, the groups of PEs are re-subdivided with the accumulated subtotals for each bin being combined into half as many PEs at each step. The subdividing process continues until there is one PE in each group and each PE has the total value for one bin from the portion of the

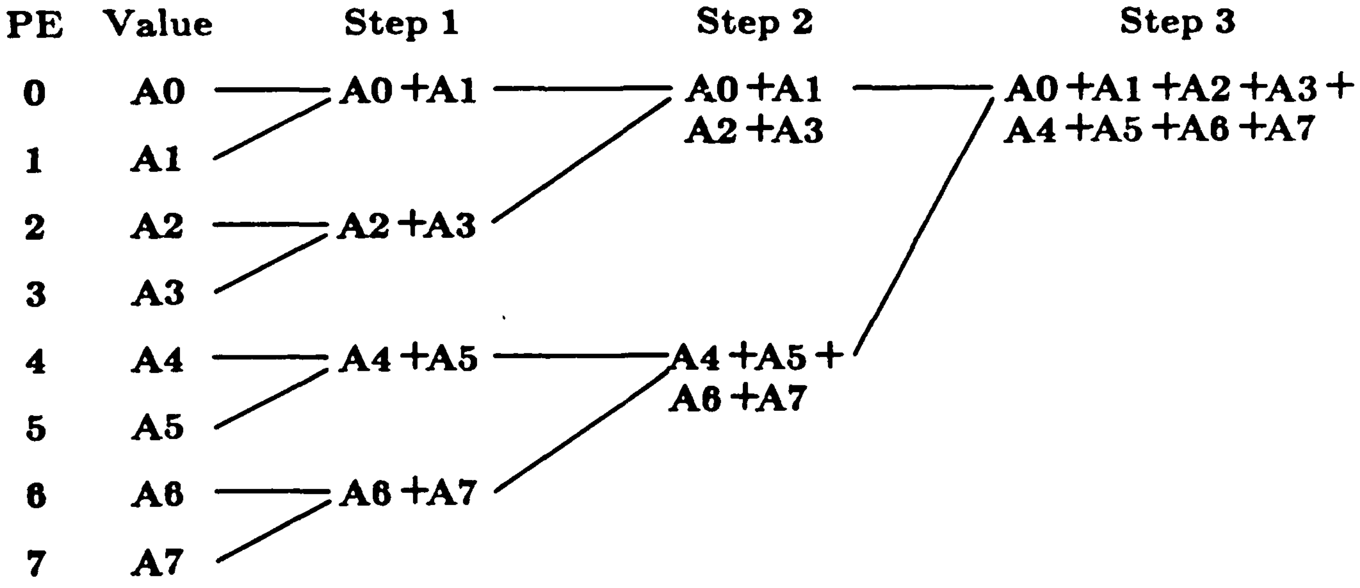


Figure 3-11. Recursive doubling with eight PEs.

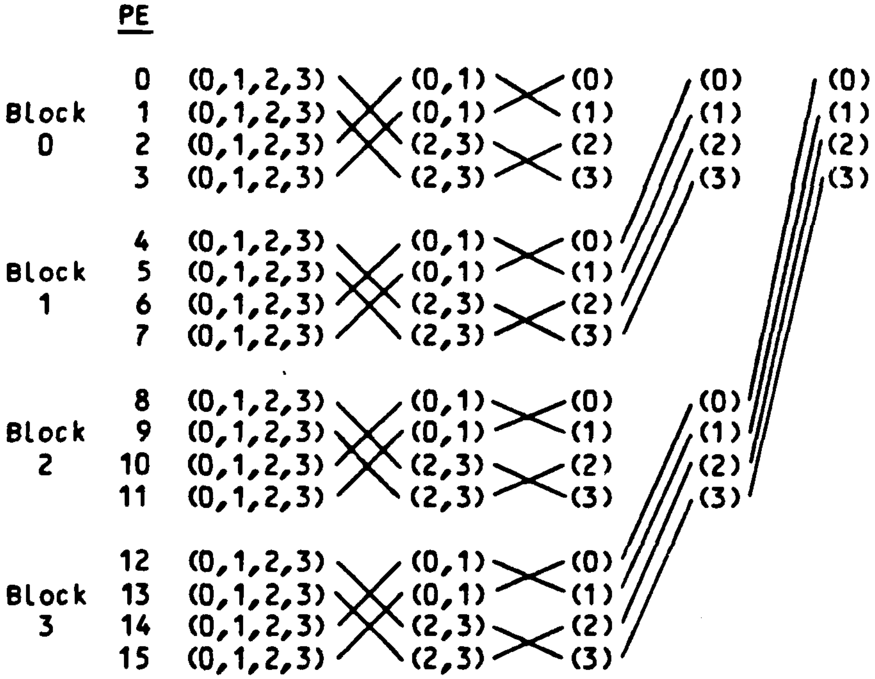


Figure 3-12. Histogram calculation for N=16 PEs, B=4 bins. (w,...,z) denotes that bins w through z of the partial histogram are in the PE.

image contained in the B PEs in its block. The next n−b steps combine the results of these blocks to yield the histogram of the entire image distributed over B PEs. The sum for bin i ends up in PE i, for $0 \leq i < B$. This is done by performing n−b steps of a recursive doubling algorithm to sum the partial histograms from the N/B blocks, shown by the last two steps of Figure 3-12. The recursive doubling steps are done for all B bins simultaneously.

A sequential algorithm to compute the histogram of an M-by-M image requires M^2 additions. The SIMD algorithm uses M^2/N additions for each PE to compute its local histogram. At step i in the merging of the partial histograms, $0 \leq i < b$, the number of parallel data transfer/adds required is $B/2^{i+1}$. This is because at step i each PE saves half of the bins it accumulated at the last step and sends the other half to another PE. For example, in Figure 3-12, when B=4, at step i=0, PEs 0, 1, 4, 5, 8, 9, 12, and 13 ("group A") send their bin 2 data, while PEs 2, 3, 6, 7, 10, 11, 14, and 15 ("group B") simultaneously send their bin 0 data. The bin data received by each PE is then added to the corresponding bin data the PE is holding. Then group A sends their bin 3 data while group B simultaneously sends their bin 1 data. Again in each PE, the received bin data is added to the corresponding bin data being held. Thus, $2 = B/2^{i+1}$ transfer/adds are used. At step i=1, (1) PEs 0, 4, 8, and 12 send their bin 1 data, (2) PEs 1, 5, 9, and 13 send their bin 0 data, (3) PEs 2, 6, 10, and 14 send their bin 3 data, and (4) PEs 3, 7, 11, and 15 send their bin 2 data, all simultaneously. The matching bin data values are then combined in all PEs. Thus, $1 = B/2^{i+1}$ transfer/add is used.

A total of B−1 $(=\sum_{i=0}^{b-1} B/2^{i+1})$ transfer/adds are therefore performed in the first b steps of the algorithm. Then n−b parallel transfers and additions are needed to combine the block histograms. These n−b steps are just B recursive doublings executed simultaneously as shown in Figure 3-12. This technique therefore requires B−1+n−b parallel transfer/add operations, plus the M^2/N additions needed to compute the local PE histograms. For example, for N=1024, M=512, and B=256, the sequential algorithm would require 262,144 additions; the parallel algorithm uses 256 addition steps plus 257 transfer/add steps. Again, both the Cube and ADM multistage networks can perform each of the required inter-PE data transfers in one pass through the network.

3.3.4 Contour Extraction

Now consider an SIMD/MIMD parallel implementation of *contour extraction* [TuA83]. Two algorithms from a contour extraction task are *edge-guided thresholding (EGT)* and *contour tracing.* The EGT algorithm is used to determine a set of optimal thresholds for quantizing the image [MiR81]. The contour tracing algorithm uses the set of thresholds to segment the image and trace the contours. It is assumed that the image to be processed is distributed among the PEs as in the smoothing example.

The EGT algorithm consists of three major steps. First, the Sobel edge operator [DuH73] is used to generate an edge image in which gray levels indicate the magnitude of the gradient. A figure of merit which indicates how well a given threshold gray level matches edges in the edge image is then computed for every possible threshold. Finally, the maximum value of the figure of merit function is chosen to determine the threshold level. This is done for each PE's subimage independently; thus, the threshold levels may differ from one subimage to the next. The window-based Sobel operator calculations and inter-PE communications used in the SIMD EGT algorithm are very similar to those discussed earlier for the image smoothing algorithm.

The MIMD contour tracing algorithm has two phases. In Phase I, PEs segment their subimages based on the threshold value each calculated using the EGT algorithm. All local contours (both closed and partial) are traced and recorded. In Phase II, partial contours traced during Phase I are connected.

In Phase I there is no PE-to-PE communication. Each PE creates a segmented subimage for a particular threshold T by assigning a value of one to subimage pixels having a grey level greater than or equal to T, and a value of zero to the others. Contour tracing begins with each PE scanning the rows of its segmented subimage beginning with the first pixel of the top row. Scanning stops when a *start point* of a new contour is found. A start point is a pixel with value one which has a zero-valued neighbor to either or both sides. Contours are traced in either a clockwise or counter-clockwise direction and the Freeman direction codes [Fre61] of the "chain" of pixels are recorded. When a pixel from an adjacent subimage would be required to determine the next direction of the contour, a point of indecision is reached. Such a point is recorded as an end point, and the algorithm returns to the start point to trace the contour in the opposite direction until another point of indecision is reached. Closed contours that are contained within a subimage are traced completely during Phase I.

In Phase II, each PE attempts to connect its partial contours to those located in neighboring PEs. PEs consider each partial contour's end point in turn and try to find a possible extending contour in a neighboring PE. Once such an extending contour is found, the process is repeated, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended. A protocol is necessary to prevent more than one PE from trying to use the same partial contour as an extending contour at the same time. Phase II is complete when all of the contours have been connected.

The contour extraction task demonstrates the advantages of several features of PASM. The local neighbor inter-PE transfers needed for the SIMD EGT algorithm can be performed by all PEs simultaneously, as was discussed for the SIMD smoothing algorithm. The global inter-PE communications for the MIMD contour tracing can be performed efficiently by either the multistage Cube or ADM networks. Lastly, the ability of the PEs to operate in either SIMD or MIMD mode allows the most appropriate type of parallelism to be employed by each algorithm in the task.

3.4 The PASM Prototype Design

3.4.1 Introduction

A prototype of the PASM system, with N=16 and Q=4, is currently being constructed (see Figure 3-13). Only off-the-shelf components are used; this eliminates the need for VLSI chips and reduces development time and construction costs. By using a modular design concept, twelve different types of physical boards are being used to construct the prototype, two of which are commercial products. These are:

1. *CPU (commercial)* -- Motorola MC68010 16-bit microprocessor, VMEbus interface, and local memory;
2. *Memory* -- VMEbus-based dual-ported arbitrated access 1-Mbyte dynamic memory with byte parity using 256K-by-1 technology;

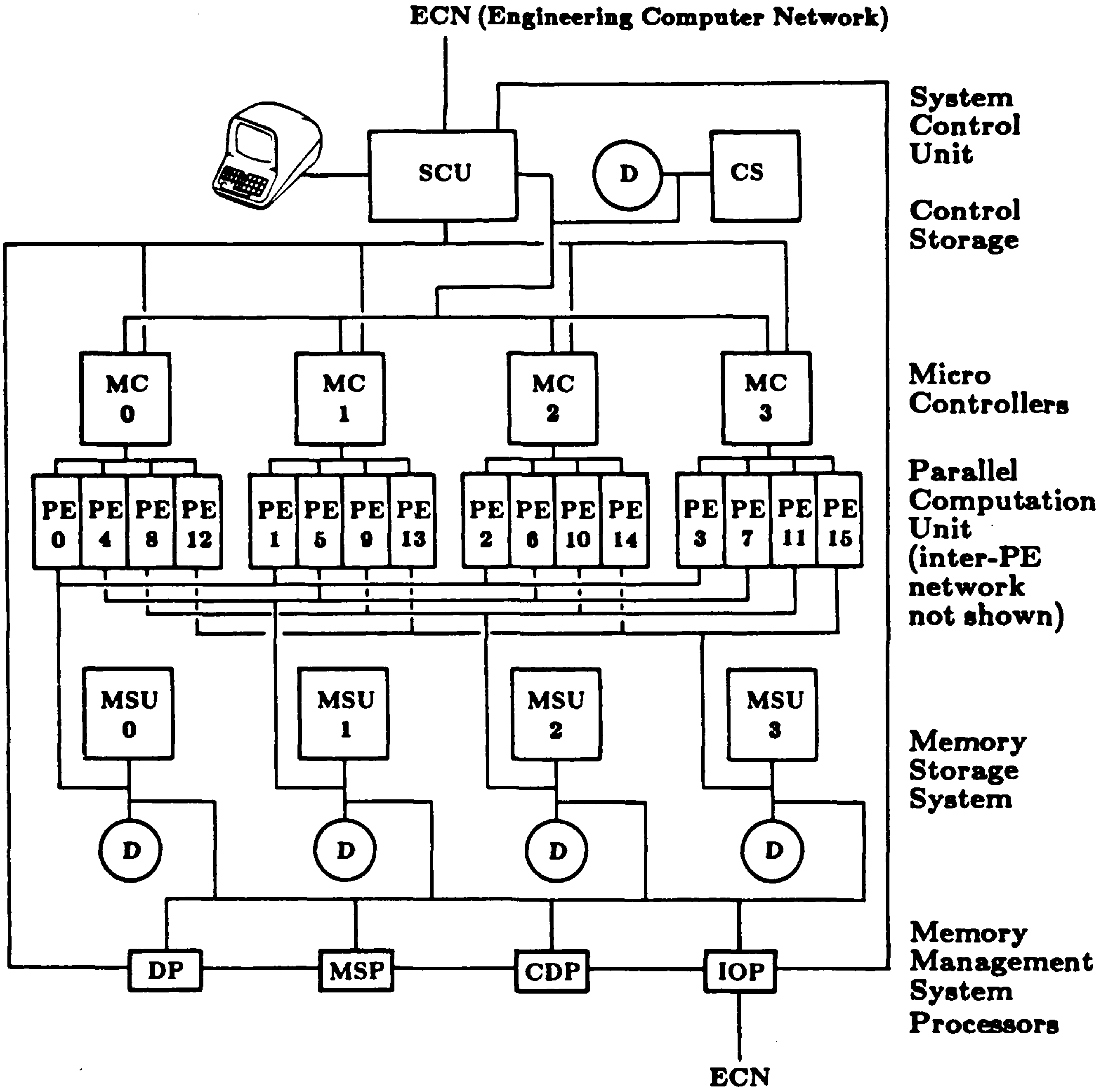


Figure 3-13. The PASM parallel processing system prototype.

3. *MC-PE I/O* -- channels for MC to PE instruction broadcast, PE enable signals, and MC-PE communication;
4. *PE-Network I/O* -- interfaces for establishing and using the multistage interconnection network;
5. *Fetch Unit* -- specialized MC unit for fetching and broadcasting SIMD instructions;
6. *Disk Controller (commercial)* -- VMEbus-based, capable of controlling up to four physical disk units, two of them Winchester-technology;
7. *Disk Access Switch* -- interfaces Control Storage or a Memory Storage Unit to MC or PE memory units;
8. *Network Extra Stage Box* -- two-by-two interchange box implementing the extra stage and extra stage bypasses of the Extra Stage Cube network;
9. *Network Intermediate Stage Box* -- two-by-two interchange box implementing the intermediate stages of the Extra Stage Cube network;
10. *Network Output Stage Box* -- two-by-two interchange box implementing the output stage and output stage bypasses of the Extra Stage Cube network;
11. *Parallel Port* -- parallel I/O channels for the connections between the System Control Unit and MC, MC and Memory Management System, etc.;
12. *Partition Controller* -- inter-MC communication and synchronization for multiple-MC partitions.

The System Control Unit, MCs, PEs, Memory Management System processors, Control Storage and Memory Storage Unit processors, and the PE interconnection network can all be implemented by using these boards in different configurations.

The Parallel Computation Unit consists of 16 PEs connected by an Extra Stage Cube interconnection network. Each group of four PEs is controlled by an MC. The prototype could be readily extended to 8 MCs and 8 PEs per MC.

The System Control Unit (SCU) for the prototype is a dedicated microprocessor responsible for the overall orchestration of the activities of the system. The Control Storage microprocessor (CS) responds to file system requests from both the SCU and the MCs. Control Storage uses a Winchester-technology secondary storage device (D). In order to allow a larger group of users to access PASM, the System Control Unit serves as a link between PASM and the Engineering Computer Network (ECN) at Purdue University. ECN is a local network connecting several dozen micro, mini, and super-mini computers. A PASM user's terminal is physically connected to an ECN host computer. The host provides the environment for the development, compilation, and debugging of SIMD and MIMD programs to prevent the System Control Unit microprocessor from being burdened. Commands (jobs) initiated by users are sent by the ECN host to the System Control Unit which schedules the jobs to be run on the parallel machine. The prototype system console is used for system startup and monitoring of PASM activities.

Mass storage for the PEs is provided by four high capacity Winchester technology disk drives (D), each controlled by a Memory Storage Unit (MSU) microprocessor. The Memory Storage Units are managed by the Memory Management System, consisting of a Directory Lookup Processor (DP), a Memory Scheduling Processor (MSP), a Command Distribution Processor (CDP), and an Input/Output and Reformatting Processor (IOP). User programs and data can be received from or sent to ECN peripherals such as additional mass storage or image input/output devices.

The complete prototype includes 30 processors. There are 16 processors for the PCU PEs, four for the MCs, four for the MSU, one for Control Storage, four for the Memory Management System, and one for the SCU.

3.4.2 PE Organization

A prototype PE consists of a CPU board, two Memory boards, an MC-PE I/O board, and a PE-Network I/O board. One "port" of each of the two 16-bit dynamic Memory boards resides on the CPU VMEbus. The other port of each board is connected to a Disk Access Switch board in a Memory Storage Unit. This implements the PASM double-buffered memory scheme.

A PE's MC-PE I/O board requests and receives SIMD instructions and enable signals from the MC and disables the PE when necessary. It does this by reserving a range of addresses known as the *SIMD Instruction Space*. When the PE CPU's program counter points into this space, it signals a request for the "next" instruction as determined by the MC. The actual program counter value within the space is irrelevant; any access within the space is treated as an SIMD instruction request. Any address outside the space is treated normally; i.e., as a reference to an (MIMD) instruction, a datum, or an I/O device in the PE's memory or I/O space. Thus to cause the PEs to switch to MIMD mode, the MC broadcasts a "jump to subroutine at A" instruction, where A is an address not in the SIMD Instruction Space. The former program counter value (in the SIMD Instruction Space) is stacked and execution begins in MIMD mode. To revert to SIMD mode, PEs execute a "return" instruction which reloads the program counter residing in the SIMD Instruction Space. MIMD programs can use this simple mechanism to quickly synchronize the PEs.

When all active PEs in a partition request an instruction in SIMD mode, an instruction word is broadcast from the MC(s) and placed on all the PE data busses simultaneously. Each PE decodes the instruction and performs the operation or requests additional operand words. Note that conventional 68000-family CPU instructions are broadcast rather than control signals as in many other SIMD machines. In MIMD mode, instructions and data are contained wholly within a PE's memory and no instructions are broadcast by the MC. Since MIMD program instructions for each PE are stored in the PE's local memory and SIMD program instructions are broadcast to the PE by its MC, the interconnection network is used solely for inter-PE data communication rather than for both inter-PE communication and instruction fetch operations.

In the prototype, a PE is disabled by intercepting instructions broadcast to it on the MC-PE I/O board. With an asynchronous bus such as the VME, a PE CPU can be prevented from "seeing" an instruction by withholding the acknowledgment signal used to indicate the end of a bus cycle. Therefore, the PE CPU "waits" until an instruction for which it is to be enabled arrives.

An MC-PE communication channel based on General Purpose Interface Bus (GPIB) technology is also found on the MC-PE I/O board. This channel is used by an MC in MIMD mode to coordinate its PE's activities. It is also used by PEs to alert their MC of an internal fault or error. A separate channel on the MC-PE I/O board is used to send the result of the evaluation of a data-condition (in a "where statement") to the MC. The set of results from all PEs in a partition forms the data conditional mask. Finally, the MC-PE I/O board contains a socket for the Motorola MC68881 Floating Point Co-processor.

The PE-Network I/O board contains parallel ports that interface the PE CPU to the interconnection network. Network paths are established and data is sent to or read from the network using conventional 68000-family CPU instructions to control the I/O ports.

PASM's use of local PE memories is appropriate for many image and speech processing algorithms because images and speech samples can be subdivided and distributed among PEs to be operated upon in parallel. However, there are algorithms for which at least some memory shared among the PEs is useful. An example is the contour tracing algorithm where partial contour tables need to be accessed by several PEs. Shared memory plays a significant role in MIMD programs: shared variables are often used for process synchronization, interprocess communication, and so on.

One way to emulate shared memory in the prototype is to designate part of each PE's address space as shared. Assuming a machine-wide shared address space of S bytes, S/N bytes of physical shared memory would reside in each PE and each PE would hold memory responding to distinct addresses. An attempt to access a shared memory location for which there was physical memory in the PE would be handled normally. However an attempt to access a shared memory location which is held by another PE results in a "bus error" trap (due to the non-existent memory) and initiation of exception processing. During the exception processing, the non-local shared address that was generated is examined and the remote PE in which it resides can be determined. A message is generated and sent through the interconnection network to the remote PE requesting the value of the data item at the shared address. The remote PE responds to the request by fetching the data from its local memory and returning it through the interconnection network. As part of the exception processing, the value returned is patched into the run-time stack and the faulted bus cycle is re-run. These actions are equivalent to those that occur in a virtual memory system when an address is found to be non-resident. The performance penalty is rather severe since exception processing is done at both the local and remote PEs and since the interconnection network is traversed twice. An enhancement to improve this situation is discussed in subsection 4.4.

3.4.3 Interconnection Network Implementation

The PASM prototype's interconnection network is a circuit-switched implementation of the Extra Stage Cube network. Circuit switching was chosen for its ease of implementation as well as its particular suitability (when compared to packet switching) for the synchronized inter-PE data transfers of complete subimage rows or columns such as those shown in Figure 3-10. Using a circuit-switched network, prior to any message transmission between a network source-destination pair, a physical path through the network must be made connecting the pair. This path is established through the use of a request-grant protocol. This connects the source-destination pair for the duration of the message transmission. Individual words within the message are transferred from the source PE to the destination PE using a handshaking protocol between the parallel ports that interface the PEs to the network.

The PASM prototype network is being constructed from readily available SSI/MSI integrated circuits. It is anticipated that a full 1024-PE system would integrate custom-designed LSI components into the network design. In its current configuration of 16 PEs, the network consists of five stages of interchange boxes with eight boxes per stage for a total of 40 Network Interchange

Box boards. A path through the network can be established in approximately 1000 ns (assuming no delays due to network conflicts). The data itself can be transmitted from a network input port to a network output port at a rate of one 16-bit word every 400 ns. With the PEs themselves operating on a 10 Mhz system clock, these transfer times are fast enough so that the network does not act as a bottleneck to the computation process under execution.

The initial prototype implementation uses the PE CPUs to perform all actions required for interconnection network transfers. Thus a CPU has to set up a path through the network, perform error checking, start and monitor watchdog timers, etc. This causes high overhead whenever data has to be transferred through the network. The impact of the overhead is increased when emulating shared memory with the approach discussed earlier.

3.4.4 The Network Interface Unit

One way to significantly enhance the performance of the interconnection network is to use a special-purpose *Network Interface Unit (NIU)* as part of each PE. An NIU is currently being designed and will replace the current prototype PE-Network I/O board. An NIU for PASM has three primary functions: it handles PE-to-PE message passing, it emulates shared memory accesses, and it acts as a sophisticated DMA controller. In order to perform these functions efficiently, the NIU contains a fast bipolar microprocessor, a CPU-NIU interface that allows the NIU to access the CPU's memory directly, a dual-ported memory (mailbox RAM) for message-passing between the PE CPU and the NIU, memory to buffer outgoing and incoming messages, a block of memory used in the emulation of shared memory, and logic interfacing the NIU to the network itself. A co-processor to handle network transfers is also used in systems such as the BBN Butterfly [CrG85] and Cm* [SwF77].

PE-to-PE message passing is performed under control of the PE CPU. When a PE needs to send a message to another PE, it places a message transfer request into the mailbox RAM and the NIU reads this request. The NIU then fetches the actual message from the CPU's main memory, sets up a path through the network, sends the message, and informs the PE CPU when the transfer is completed. In the meantime, the PE CPU can perform other processing tasks thus overlapping network I/O and processing. When a message arrives at a destination, the NIU there accepts it and informs its PE CPU that a message is available. The PE CPU provides the NIU with a destination buffer address for the message and the NIU stores it there.

The NIU facilitates complex message schemes. A simple way to specify a message is to give the NIU the location and size of a buffer in memory and to instruct it to transfer it to a destination PE. However, since the NIU processor is general-purpose, it can be programmed to move more complicated data structures. For example, a message can be specified by a starting address, size, stride, and count. Here, the NIU transfers "count" data items of the given "size" beginning at the starting address and incrementing the address by "stride" to obtain each new item.

If desired, the NIU can buffer a message and use high speed DMA transfers from a source buffer via the network into a destination buffer. This way the time required to perform the address calculations for fetching data from the CPU memory does not slow the actual network transfer, thus effectively reducing network traffic by decreasing the average time a network path stays established.

Through the use of an NIU, shared memory can be implemented with less performance penalty than the software emulation scheme described earlier. For the prototype, an NIU is being designed and constructed that contains 2K bytes of shared memory and is associated with each PE. This results in a total shared memory size of 16*2K bytes = 32 K bytes. Whenever a PE CPU accesses any location in the shared memory space, the access is decoded by the local NIU. If the access was directed to the shared memory slice located in the local NIU, the NIU performs the access on the memory without having to use the interconnection network. Otherwise, the NIU determines the PE in which the slice is located, sets up a path to the proper remote PE, and sends a message to the remote PE requesting shared memory access. This message is handled not by the remote PE CPU but exclusively by the remote NIU. If the message specifies a write to shared memory, the remote NIU writes the data into its shared space. If it was a read access, the remote NIU sets up a path to the source PE, and passes the value obtained from the shared space to it. Using this scheme, processing in the remote PE CPU is not interrupted and the fast NIU processor results in high speed accesses.

3.4.5 MC Organization

An MC consists of a CPU board, two Memory boards, an MC-PE I/O board, a Fetch Unit board, and a Parallel Port board. The MC CPU coordinates its PEs in MIMD mode; in SIMD mode, it performs the program flow operations (such as loop counting and branching) and the masking operations. One port of each of the Memory boards is on the MC CPU's VMEbus while the other is attached to a Disk Access Switch board in Control Storage. The MC-

PE I/O board is the same one used in the PEs, but populated with somewhat different chips. Its main functions in the MC are to control the GPIB channel, to accept N/Q bits of a data conditional mask from its PEs, and to interface the set of inter-MC result and synchronization buses used for evaluation of the "if any"-type conditions. The Parallel Port board is used to communicate with the System Control Unit and with the Memory Management System.

A 68000-family MC CPU alone cannot fetch SIMD instructions, decode them, execute the control instructions, and broadcast the arithmetic instructions to the PEs at a rate fast enough to avoid "starving" the PE CPUs of instructions. One of the specialized components of the MC that performs some of these functions is the Fetch Unit which is controlled by the MC CPU. The Fetch Unit consists of a high-speed dual-ported Fetch Unit Memory, a finite state machine controller, and a FIFO buffer. SIMD PE instructions are placed in Fetch Unit Memory while MC control instructions are placed in the dynamic MC CPU memories. When a block of one or more PE instructions is to be broadcast to the PEs, the MC CPU writes a special command word to the Fetch Unit controller indicating the base address and the length of the block. The Fetch Unit Controller fetches the words of the block one-by-one from Fetch Unit Memory and places them in the FIFO buffer in preparation for broadcast. As each instruction is placed in the FIFO, the enable signals controlling which PEs are to be enabled for that instruction are also enqueued in the FIFO. The MC CPU can also create instructions at execution time and place them in the FIFO. This would be done if the MC needed to broadcast some run-time-dependent value to each of the PEs at a certain point in a program.

The Fetch Unit Controller fills the FIFO buffer while the PEs drain it. When all of the PEs associated with an MC request an instruction, those requests are combined by the Partition Controller board. Only when all PEs in a partition request an instruction, one is dequeued from the FIFO and broadcast to them. The FIFO buffer allows overlap between the operations of an MC and its PEs in SIMD mode, thereby improving performance.

3.4.6 Secondary Memory Systems

Control Storage and the Memory Storage Units each consist of a CPU board, two Memory boards, a Disk Controller board (and physical disks), one or more Parallel Port boards providing channels for incoming file system requests, and a Disk Access Switch board for each MC or PE memory it serves. Memory Management System processors each consist of a CPU board and a number of Parallel Port boards. The CPU boards lie on a common VMEbus with a shared dynamic memory allowing easy exchange of shared data structures. They communicate control information over the parallel port channels which do not require VMEbus access.

3.5 PASM Languages and Operating System

3.5.1 Languages

MIMD programs for PASM can be written by coding each instruction stream in a conventional serial programming language and using calls to operating system functions to allow the streams to communicate. Even if higher-level languages with embedded multi-tasking primitives are used (e.g, Ada, CSP), the compiler produces a number of sequential instruction streams which can be run in parallel and which communicate and synchronize via operating system calls. Either approach can be used for PASM, although only the first approach is supported by the C compiler and 68000-family assembler developed for use by the PASM prototype. The MIMD-specific operating system functions called in MIMD programs are being developed as part of the PASM operating system.

SIMD mode programming requires a different type of language since the single instruction stream controls a machine consisting of a scalar unit (the control unit) and a variable-length vector unit (the PEs). A prototype-specific SIMD assembly language has been implemented. In this language, instructions that are to be executed by the 68000-family MC CPU have their assembly language mnemonics prefixed by a "c_" (for control unit), while instructions to be broadcast to the PEs are prefixed with a "p_." The SIMD program is coded in a single stream with the control and PE instructions intermixed. The assembler separates the control and PE instructions so that they can be placed in separate memories at execution time. Where blocks of PE instructions are removed to separate them from the control stream, Fetch Unit control words are inserted to effect the broadcast of the PE instructions at execution time. Programs written in this assembly language have been run on a PASM simulator. In addition, SIMD extensions to the C and Ada programming languages have been proposed by members of the PASM development group. A parallel implementation of a LISP interpreter for use in image understanding applications has also been developed and simulated.

3.5.2 Operating System

The PASM operating system is distributed among all of the PASM CPUs. It is logically divided into two layers: the *local kernel* layer and the *PASMOS* (PASM Operating System) layer. A local kernel is resident on all CPU boards and is responsible for local memory management, local process scheduling and synchronization, local I/O device control, local file operations, and reliable communication with other CPUs. All local kernels are identical except for their hardware dependencies; e.g., the physical locations of memory and I/O devices. PASMOS is a collection of operating system routines that are distributed among the PASM components. For example, the PASMOS routines for choosing the partition on which a job is to be run exist only in the System Control Unit. On the other hand, PASMOS routines for handling file load/unload requests from the PEs exist on the MCs, the processors of the Memory Management System, and on the Memory Storage Units.

Shared resources and activities that involve the control of multiple CPUs are the domain of PASMOS. The PASMOS routines use the facilities of the local kernels to perform their functions. User programs make calls to operating system functions in both layers. For example, a request for a PE data file loading operation in SIMD mode results in simultaneous calls by all PEs to their own local kernel memory allocation routines to determine where the data should be placed. The synchronization of the request (to determine when all of the files have been loaded) is a PASMOS process; however, local kernel I/O drivers are used to perform the low-level communication of the request itself. Development of the PASM prototype's operating system is underway.

PASMOS makes scheduling decisions based on processor availability and task priorities. However, it is incapable of automatically making intelligent decisions about the "best" parallel algorithm to use for a particular data set or for highest efficiency given the number of processors available. For example, several algorithms may be available to do a single task: one SIMD, one MIMD, one very fast but suitable only if a large number of PEs can be assigned to it, one that is slow but does the task particularly "well," and so on. The powerful reconfiguration capabilities of PASM can be exploited by an *Intelligent Image Understanding System* that is programmed to take best advantage of the machine [DeS85]. Such a system currently under study would be instructed to perform a task consisting of many subtasks. It would automatic.lly select the best algorithm to execute each of the subtasks from an algorithm database and schedule these subtasks so that an optimum execution time is achieved. A precedence graph of subtasks would be employed to show the interdependence of the operations to be performed. Decisions would be based upon the current system state (i.e., available PEs, free memory, state of the subtasks) and are continually updated as the system state changes. Thus the image understanding system is capable of adapting to changing requirements. For example, it can concentrate computing power on a certain part of an image if it is determined that this part is of particular interest and it can concentrate computing power on a particular subtask to speed its execution.

3.6 Summary

This chapter provided an overview of the PASM design concepts, the PASM prototype, and examples of parallel image processing algorithms. The PASM design is for a large-scale, dynamically reconfigurable, SIMD/MIMD parallel processing system. The system design was developed as a result of simulation studies and parallel algorithm analyses. Thus, both the system hardware and software were application driven. A 30-processor prototype is currently under construction. Table 1 summarizes the PASM design parameters. The rest of this section summarizes some of the PASM design features.

The possible advantages of a reconfigurable system such as PASM include:

(a) *fault tolerance* -- If a single PE fails, only those machine partitions which must include the failed PE need to be disabled. The rest of the system can continue to function.

(b) *multiple simultaneous users* -- Since there can be multiple independent machine partitions, there can be multiple simultaneous users of the system, each executing a different program.

(c) *program development* -- Rather than trying to debug a program on, for example, 1024 PEs, it can be debugged on a smaller size partition of 32 PEs.

(d) *variable machine size for efficiency* -- If a task requires only N/2 of N available PEs, the other N/2 can be used for another task.

(e) *subtask parallelism* -- Two independent subtasks that are part of the same job can be executed in parallel, sharing results if necessary.

(f) *multiple processing modes* -- An algorithm can be executed using either the SIMD or MIMD mode of parallelism, whichever is more efficient. A task requiring algorithms that use both modes of parallelism can be executed using the same set of PEs.

The advantageous features of both the multistage Cube and the ADM networks include:

(a) up to N simultaneous transfers are possible;

(b) they are partitionable into independent subnetworks;

(c) they can be controlled in a distributed fashion using routing tags;

(d) one PE can broadcast to all or a subset of the others;

(e) there are a variety of implementation options for these networks;

(f) they can be used in SIMD and/or MIMD operations; and

(g) they can support efficient global as well as local (nearest neighbor) inter-PE communications.

An additional advantage of the Extra Stage Cube is that it is single-fault tolerant.

Permanently assigning a fixed number of PEs to each MC has several advantages over allowing a varying assignment such as used in MAP [Nut77]:

(a) *scheduling* -- The operating system need only schedule (and monitor the "busy" status of) Q MCs, rather than N PEs (when Q=32 and N=1024, this is a substantial savings).

(b) *hardware simplicity* -- No crossbar switch is needed for connecting PEs and control units (such as proposed for MAP [Nut77]).

(c) *software simplicity* -- There is no need to do the bookkeeping of recording PE to MC assignments.

(d) *network partitioning* -- The fixed assignment supports network partitioning.

(e) *secondary storage* -- The fixed assignment allows the efficient use of multiple secondary storage devices.

The main disadvantage of this approach is that the size of each partition must be a power of two, with a minimum value of N/Q. However, for PASM's intended experimental environment, flexibility at "reasonable" cost is the goal, not maximum PE utilization.

The Memory Storage System design allows the loading/unloading of a machine partition of RN/Q PEs in R parallel block moves. The double-buffered PE memory modules permit this loading/unloading to be overlapped with PE execution. Similarly, the double-buffered MC memory modules allow the loading/unloading to be overlapped with execution. The Memory Management System, which coordinates these memory transfers, is implemented with multiple processors, providing parallelism at another level.

The PASM operating system functions are distributed over various system components to prevent the System Control Unit from being a bottleneck. A parallel assembly language for 68000-family processors exists, and a parallel C language for PASM is under development.

In conclusion, the objective of the PASM design is to achieve a system which attains a compromise between flexibility and cost-effectiveness for a specific problem domain. A dynamically reconfigurable system such as PASM is a valuable tool for both image understanding and parallel processing research.

CHAPTER 4

PASM PROTOTYPE DESIGN AND IMPLEMENTATION

The concept of the PASM Parallel Processing System was first published in [SiM78]. In 1984, funding became available to build a prototype of the system, and construction was begun in the Spring of 1985. This author has guided the development of the PASM prototype from paper design to working hardware. Apart from being heavily involved in the actual implementation, supervising a PASM team of as many as fifteen persons, and managing the PASM budget, the author has been responsible for many fundamental design decisions and new hardware concepts.

At the beginning of the prototype development, the basic PASM structure and some design details were fixed. Most components, however, were given as block diagrams, and many of those were rather vague. For example consider inter-processor communication. In the early design phases it was specified which PASM processors need to communicate with each other, but the type of communication channel was undefined. As part of this research effort, the communication needs have been analyzed, and all channels were specified.

In this work, many components unique to parallel processing systems were developed. The design tradeoffs for these components were analyzed with respect to particular requirements generated by the dynamically reconfigurable SIMD/MIMD structure of PASM. The Fetch Unit is a prime example. While it was specified that the Fetch Unit had to move SIMD instructions from a memory to PEs via an instruction queue, all internal components had to be defined and designed. Great care had to be taken to ensure that the hardware would work with parallel processing software. For instance, an MC must be interruptible at any given time to respond to error conditions. This requirement resulted in the addition of certain hardware components and also in the development of a strict protocol when switching from SIMD to MIMD programs. Similar considerations were made in many hardware components so that the design process was not straightforward but interactive: a new design had to be analyzed in terms of suitability for the special parallel processing application. An example is the Dynamic Memory Board. Dynamic memory chips need to be "refreshed" in short intervals. If the memory refresh in the PEs would occur at random, SIMD performance would be degraded severly. This is because PEs in a partition execute an instruction only when all PEs of the partition are ready to execute it. If the refresh occurs at random, some PE in a large partition would always be refreshing. Thus the dynamic memory refresh in the system had to be synchronized.

The components of the PASM prototype had to be analyzed to determine how they could be implemented in a simple fashion without disturbing the architectural integrity of the system. Many components were greatly simplified as compared to the original design. As an example consider the MC. Two components of the original MC were the Masking Operations Unit and the PE Address Mask Decoder. The Masking Operations Unit was intended to be controlled by a bit-slice microprocessor and would have contained a large number of components. Analysis showed that all operations the Masking Operations Unit was to perform could be easily emulated by the MC CPU. While this reduces performance slightly, the Masking Operations Unit was not implemented since it was no longer essential for a working prototype. Due to the modular structure of the prototype, the Masking Operations Unit can be added later if performance measurements show that its omission resulted in a too severe performance penalty. The PE Address Mask Decoder was originally intended to be implemented in hardware, but it was shown that it can be replaced by a table look-up without any loss of performance.

Another example is the double-buffering of the Fetch Unit memory. Implementation constraints made it problematic to design a physically double buffered memory. As discussed in section 4.3.3, the method of logical double buffering was devised to simplify the design.

A feasible physical layout of the system had to be devised. The problem of interconnecting and constructing a large system like the prototype is not trivial; in complexity (and speed) it is comparable to the fastest currently available super-minicomputers. An examples is the interconnection network that requires 40 boards to be connected in a complex pattern. The connections were examined and an optimum arrangement of boards was found that minimizes cabling complexity.

In summary, the challenge in the prototype design was to preserve the basic PASM architectural ideas in order for the prototype to be able to prove the validity of the PASM concepts, while developing a prototype design that was buildable with limited funding in a university environment. The detailed design and implementation of the PASM prototype is on the forefront of parallel computer system technology, which necessitated the use of new ideas and approaches for the prototype construction.

In Chapter 3, the PASM system and its prototype were overviewed. In this chapter, details of the hardware design will be presented. Following a section that discusses problems of building a large system like the PASM prototype in the given university environment, the advantages of a modular design for the prototype are demonstrated, the types of boards used in the prototype are discussed, and it is shown how these boards are used to build the various processors. Then the interactions between processors are shown, followed by a detailed description cf hardware design of each board that was developed in-house. The chapter concludes with a discussion of the mechanical construction of the prototype, focusing on cooling and power distribution.

4.1 Influences of the University Environment

The design and construction of a complex system like the PASM prototype in a university environment poses unique challenges and demands. The two major factors that attribute to these problems are the limited funding that was available for the construction of the prototype, and the manpower constraints that are largely unique to the university environment (one technician plus student labor).

First consider what had to be accomplished with the available funds. The foremost goal was the construction of the PASM prototype. Thus electronic, electric, and mechanical parts had to be purchased. In addition, no laboratory equipment was available when the design started, and it had to be purchased as well. This is all complicated by the rapidly changing technology and pricing structure of components.

During the PASM prototype design and construction, most work had to be accomplished by students. Most of the student work was done as undergraduate or graduate projects under the supervision of a professor or a Ph. D. student. Students for such projects have to be attracted, then they have to be made familiar with the technical issues. Only after such an initial period can they start on design projects. This process is especially time consuming during projects dealing with printed circuit board design. The students not only have to understand the design they are to layout but also have to familiarize themselves with the graphic system used for layouts. Though quality of student projects can be very high, there is no guarantee. Some designs were delayed more than two semesters due to ineffective projects. The other important drawback of student projects is the missing continuity. After the semester has ended and the student has received his grade, he will no longer work for the project, and the experience and know-how acquired during the duration of the project will be lost. During the last half year of construction, a number of students who had completed a project in the previous semester stayed with the PASM team. A significant improvement in the speed of progress could be noted.

Missing technical support staff is another issue. In a physically big system like PASM, many mechanical and electromechanical duties have to be performed. They range from soldering and wire-wrapping to installing cabling, power supplies, and fans. It is difficult to motivate students to perform such tasks efficiently if no financial incentive is given. Once money for hiring student help for such tasks was allocated, much better progress was made.

During the construction of the prototype, no serious problems were encountered. Some designs had to be redone due to fatal flaws, but all of these flaws were discovered in a relatively early stage so that no serious misinvestments in parts or labor were made. Most student projects and paid student help achieved the desired results, and no single project delayed the prototype construction. Most of the delays resulted from underestimates of specific tasks.

The best example is the mechanical construction and power distribution. In terms of man-hours, twice the time originally estimated was required to complete the task. Such delays were of course no surprise since the construction of the prototype was the first "big" hardware project for everybody involved. Since the hardware is expected to be completed during the summer of 1986, the hardware construction of the prototype will have been accomplished in under two years, and that ain't bad.

4.2 Modularity of the Prototype

In the previous section, some influences of the university environment on the PASM prototype design and construction were discussed. In this section, technical tradeoffs that in part are decided by university constraints will be described.

A fundamental issue in the prototype design is the number of processing elements in the system, and the performance each PE should achieve. The cost of a parallel processing system increases with the number of PEs, but not linearly. For example, the number of components in PASM's Extra-stage Cube interconnection network increases with $N \log_2 N$, and so does the cost. On the other hand, the SCU and the Memory Management System are identical for systems with a widely varying number of PEs, and thus give a certain constant cost component. With the given funding, a 16-PE system seemed possible to construct, and the actual implementation has shown this assumption to be valid.

Although one goal in building any computer, and especially in building parallel processing systems, is performance, this goal is not overriding when building a prototype. The PASM prototype will be a research tool in studying parallel processing, and will be used to prove the validity of the design concept. High performance would be achieved by a full system as described in [KuS85]. In the current prototype, high performance took second place to completing the system in a given two-year time frame. This avoids some of the the problems incurred during the construction of the ILLIAC IV system, which pushed not only concepts but also performance and consequentially suffered huge cost overruns and was not completed as originally conceived [Slo81].

Therefore it was decided to use complete microprocessors in the PASM prototype instead of bit-sliced CPUs which would be faster but are difficult to implement and program. Cost reasons ruled out any custom-chip solution in the system, so commercially available microprocessors and peripherals were the only ones considered. The interconnection network, a prime candidate for a custom-chip implementation, was designed with TTL logic.

Earlier studies [KuS82] have demonstrated the suitability of the Motorola MC68000 microprocessor as the CPU in a PE. At the beginning of the PASM prototype design, the MC68000 was the highest-performance single-chip microprocessor available. The MC68010 became available early in the design process. Since it is pin-compatible with the MC68000 and has slightly higher performance, it replaced the MC68000. The 32-bit version of the MC68000, the MC68020, became available too late to be considered for inclusion.

Important tradeoffs were also made on a board level. As shown in [KuS85], a physically small processing element (for example, a PE consisting of a single board) is essential when building a large system with hundreds or thousands of PEs. The design of such a small PE poses many problems: all components of the PE have to be designed at the same time, and they all have to work in order to get a working PE. A PE that consists of a number of boards, each of which has a different functionality, has many advantages in a prototype. The boards can be developed, laid out and tested individually, and manufacturing can be done in parallel. Especiaiiy important for a prototype is the flexibility gained through modularity since parts of the system can be replaced by more sophisticated or simpler components. For example, the passive network interface of the prototype could be replaced by a intelligent network interface as described in Chapter 5, without the need for modifications of other elements in the system.

Another advantage of the modularity is the possibility to use a board in more than one system component. The same CPU Board, for example, is used in every processor of the system. Standardization is another measure taken to speed up and simplify the prototype design. Originally it was intended to develop all components of the PEs (CPU Board, memory board, etc.) in-house since it was assumed that special parallel processing functions (e.g., SIMD instruction broadcast) could not be implemented with commercial product. Careful analysis showed, however, that no hardware required for parallel processing was needed on the CPU Board, and thus commercial CPU Boards were examined. Since the CPU type (MC68000 family) had been chosen early in the design phase, VMEbus-based CPU Boards manufactured by Motorola were chosen, and all boards developed or purchased subsequently are VMEbus-based. This has since proven to be a very good decision; several reasons can be identified.

The VMEbus is a commercially available bus, and thus very well defined. This definition is not restricted to standard connectors and the location of signals on the bus, but also describes the exact timing relationships between bus signals. A board designed according to these specifications will be able to interact with any VMEbus board. Also, the avenue of commercial boards is opened. A wide variety of such boards can be purchased, ranging from backplanes to CPU Boards. Backplane design is often neglected; after all, a backplane just connects connectors with each other. When analyzing the problem thoroughly, however, many issues from transmission lines to power distribution appear. Other commercial boards are even more difficult to design, for example disk controller boards. The following commercial boards were purchased:

- *MVME 110 CPU Boards* -- Motorola MC68010 16-bit microprocessor, VMEbus interface, and local memory;
- *MVME 320 Disk Controller Boards* -- VMEbus-based, capable of controlling up to four physical disk units, two of them Winchester-technology;
- *MVME 400 dual serial port boards* -- serial ports to establish connection to ECN from SCU and IOP;
- *VMEbus Backplanes* -- five, six, and ten-slot backplanes that supply power to and interconnect VME boards;

This are, of course, not all components of a parallel processing system. The following boards had to be developed in-house:

- *Disk Access Switch Board* -- interfaces Control Storage or a Memory Storage Unit to MC or PE memory units;
- *Dynamic Memory Board* -- VMEbus-based dual-ported arbitrated access 1M byte dynamic memory with byte parity using 256K bit memory chips;
- *MC-PE I/O Board* -- channels for MC to PE instruction broadcast, PE enable signals, MC-PE communication, performance timers, and floating-point support;

- *Network I/O Board* -- interfaces connecting PEs to the multistage interconnection network;
- *Fetch Unit* -- specialized MC unit for fetching and broadcasting SIMD instructions;
- *Network Extra Stage Box Board* -- two-by-two interchange box implementing the extra stage and extra stage bypasses of the Extra Stage Cube network;
- *Network Intermediate Stage Box Board* -- two-by-two interchange box implementing the intermediate stages of the Extra Stage Cube network;
- *Network Output Stage Box Board* -- two-by-two interchange box implementing the output stage and output stage bypasses of the Extra Stage Cube network;
- *Parallel Port Board* -- parallel I/O channels for the connections between the System Control Unit and MC, MC and Memory Management System, etc.;
- *Partition Controller Board* -- inter-MC communication and synchronization for multiple-MC partitions.

The System Control Unit, MCs, PEs, Memory Management System processors, I/O Processor, Control Storage and Memory Storage Unit processors, and the PE interconnection network can all be implemented using these boards in different configurations. The *System Control Unit* (Figure 4-1) consists of a CPU Board, two Dynamic Memory Boards, an SCU I/O Board, and two Parallel Port Boards which connect to the MCs, the IOP, and the DP. *Control Storage* (Figure 4-2) is composed of a CPU Board, a Disk Controller Board with an associated 65M byte winchester disk and two floppy disks, two Dynamic Memory Boards, and five Disk Access Switch Boards which are connected to the dynamic memories of the MCs and the SCU. The *Micro Controllers* (Figure 4-3) consist of a CPU Board, two Dynamic Memory Boards, a Fetch Unit, an MCPE I/O Board, and a Parallel Port Board for communication with the SCU. The *PEs* (Figure 4-4) are composed of a CPU Board, two Dynamic Memory Boards, an MCPE I/O Board, and a Network Interface Board.

The Secondary Memory System has three separate components. The *MSUs* (Figure 4-5) each contain a CPU Board, Disk Controller Board and an associated 65M byte winchester disk, two Dynamic Memory Boards, and four Disk Access Switch Boards that are connected to the memories in the PEs. The *I/O Processor* (Figure 4-6) consists of a CPU Board, two Dynamic Memory Boards, five Disk Access Switch Boards, and two Parallel Port Boards for communication with SCU and CDP. The *Memory Management System* (Figure 4-7) contains three CPU Boards implementing the *Directory Processor (DP), Memory Scheduling Processor (MSP),* and *Command Distribution Processor (CDP),* which share a common VMEbus with two Dynamic Memory Boards. Each CPU Board has Parallel Port Boards for communication with various system processors.

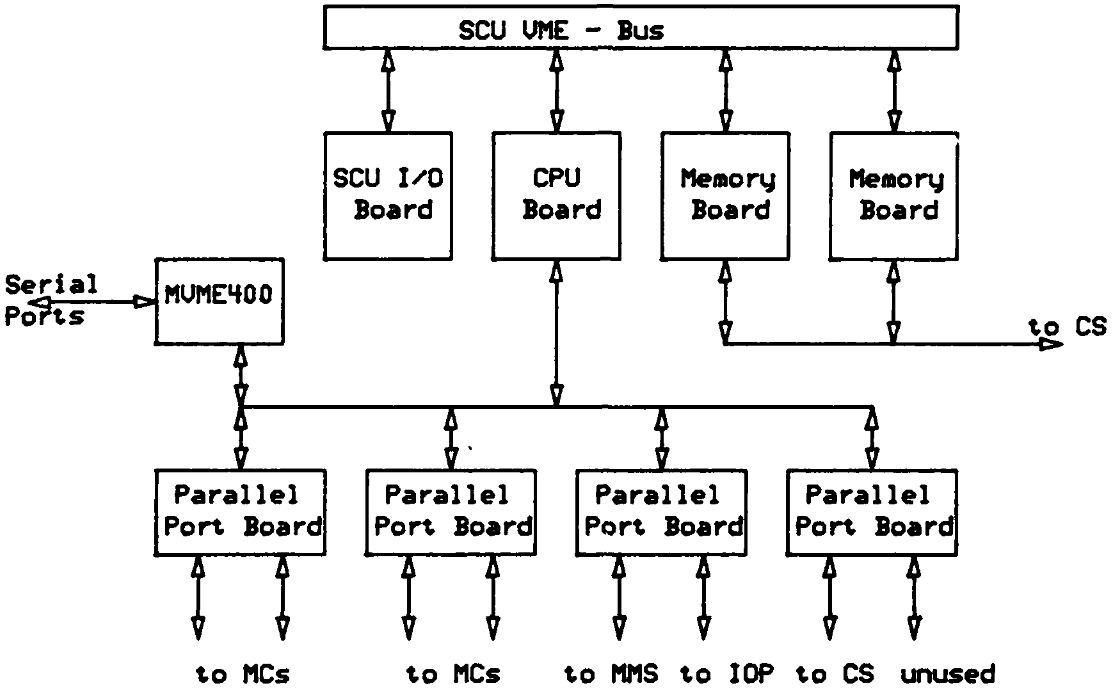


Figure 4-1. Block diagram of the System Control Unit (SCU).

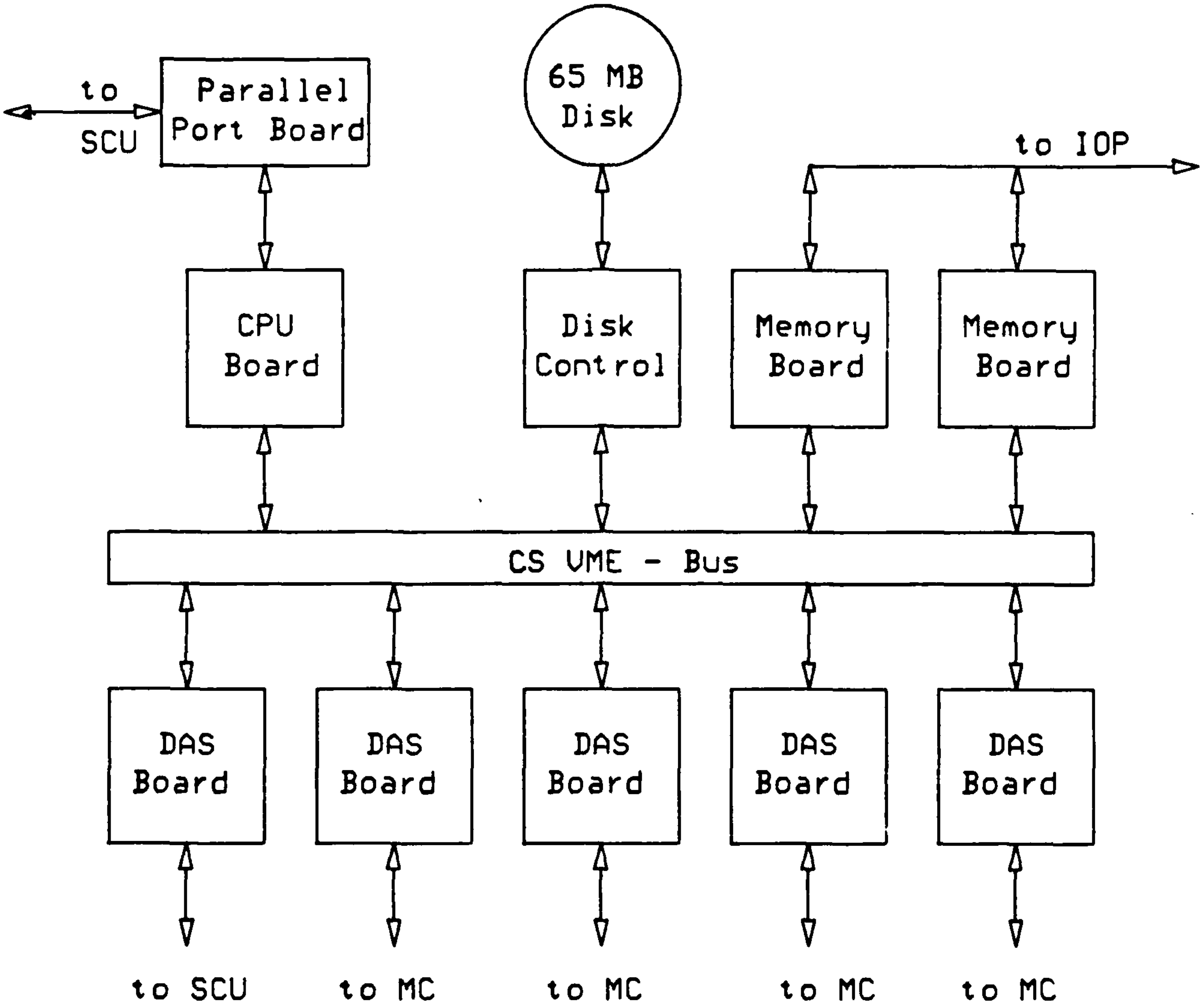


Figure 4-2. Block diagram of Control Storage (CS).

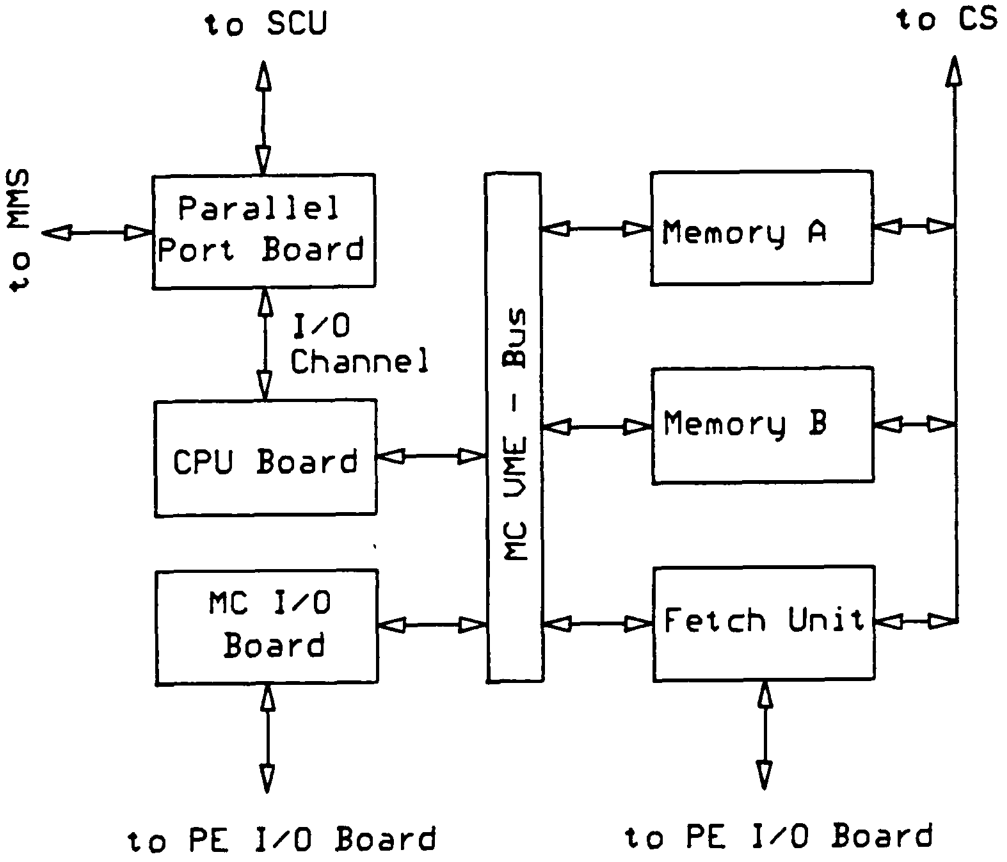


Figure 4-3. Block diagram of a Micro Controller (MC).

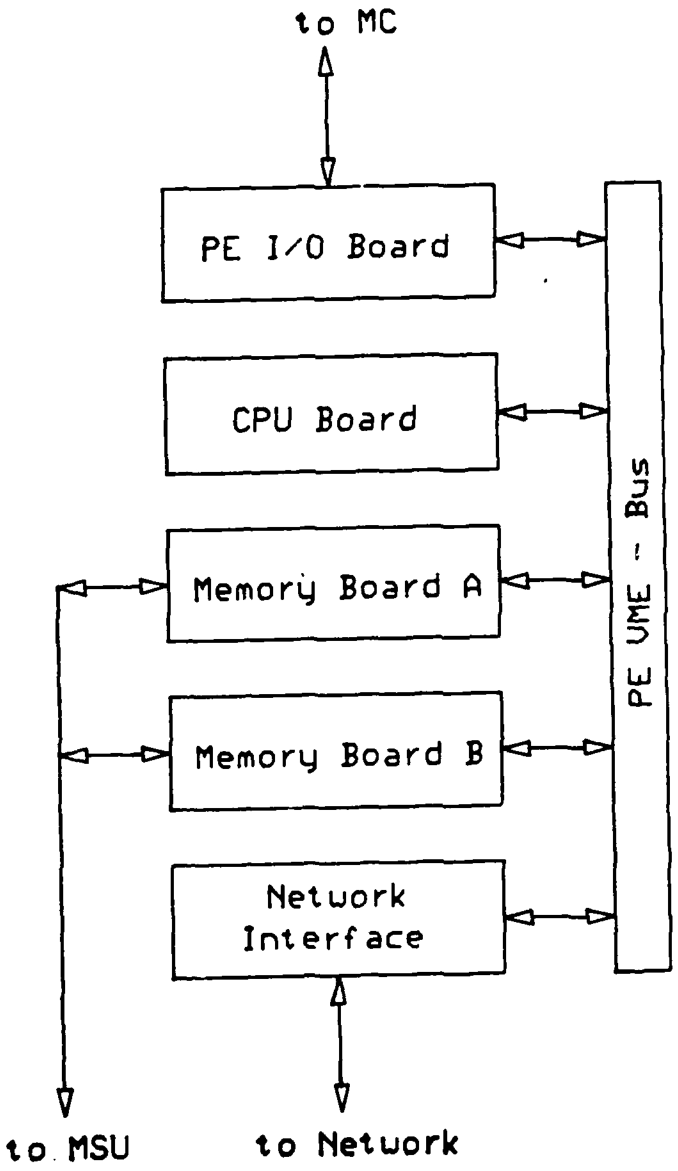


Figure 4-4. Block diagram of a Processing Element (PE).

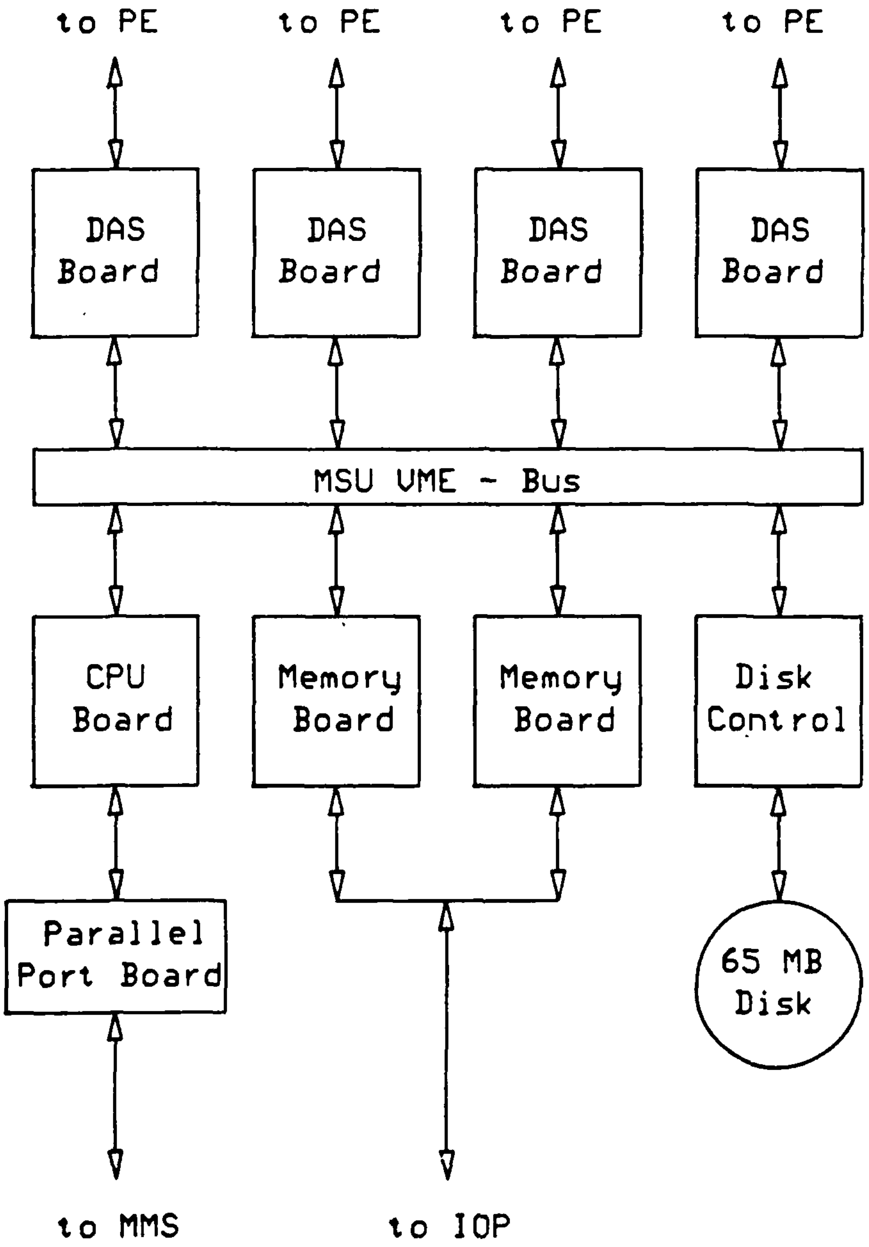


Figure 4-5. Block diagram of a Memory Storage Unit (MSU).

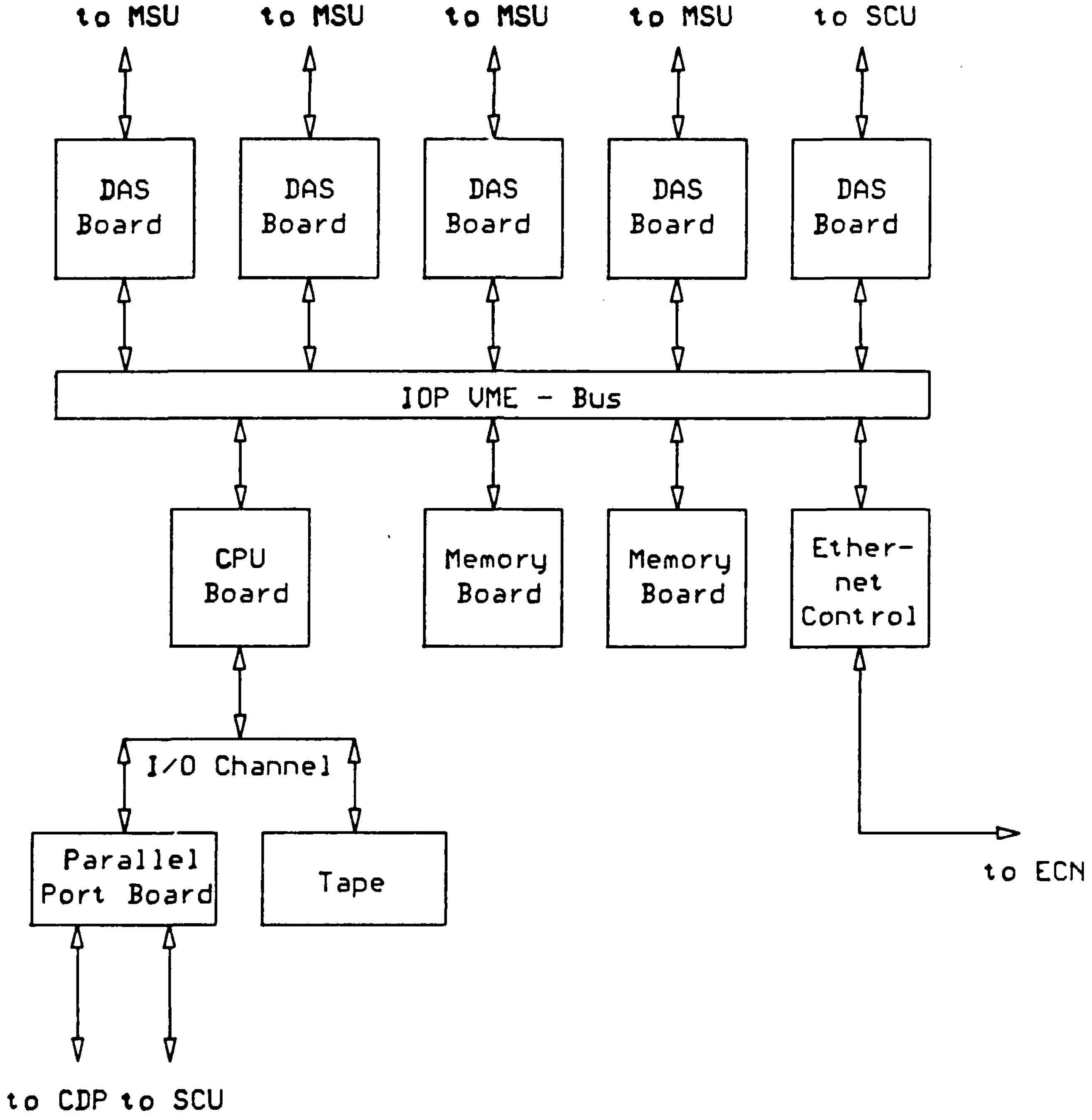


Figure 4-6. Block diagram of the I/O Processor (IOP).

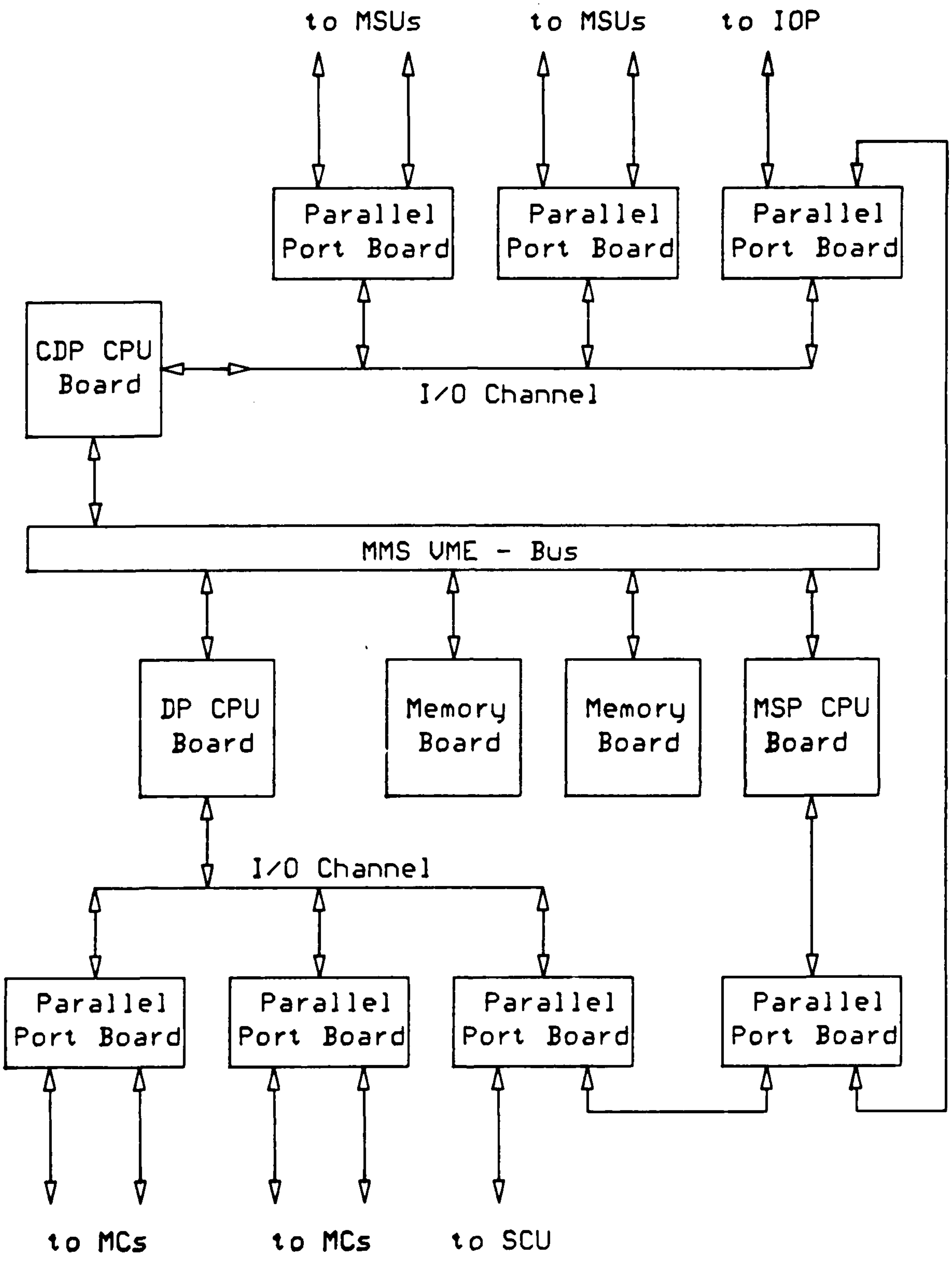


Figure 4-7. Block diagram of the Memory Management System (MMS).

Figure 4-8 illustrates the interconnections between the individual PASM components. The Extra-Stage Cube Interconnection Network is not shown explicitly. Five types of connections can be identified:

- *DAS Channels* -- These channels connect a DAS Board to the second port of a Dynamic Memory Board. The arrows in Figure 4-8 always point towards the memories. Thus, CS has access to the SCU and MC memories, the IOP has access to the MSU and CS memories, and the MSUs have access to PE memories.
- *Parallel Port connections* -- These connections facilitate bidirectional data transfer between various processor pairs. A connection can be established by either processor of the pair. The connections are used to transfer control information. For example, an MC can use it to signal the SCU the completion of a job, or it can use it to request file service from the DP.
- *MC/PE Bus* -- This connection between an MC and its associated PEs has a number of functions. It contains the SIMD Instruction Broadcast Bus that is used to send instructions to the PEs in SIMD mode; the MC/PE Communication Channel that is used by MC and PEs to exchange control information; and the Condition Code Bus that informs the MC of the result of data conditional operations in the PEs.
- *Shared MMS Bus* -- The three Memory Management System Processors (DP, MSP, and CDP) share a common VMEbus and have shared main memory.
- *Serial Link* -- PASM's system console is conneced to the SCU via a serial link, and in the initial prototype implementation SCU and IOP have a serial link to the Engineering Computer Network (ECN). The serial links to ECN could be replaced by high-speed ethernet links if funding is available.

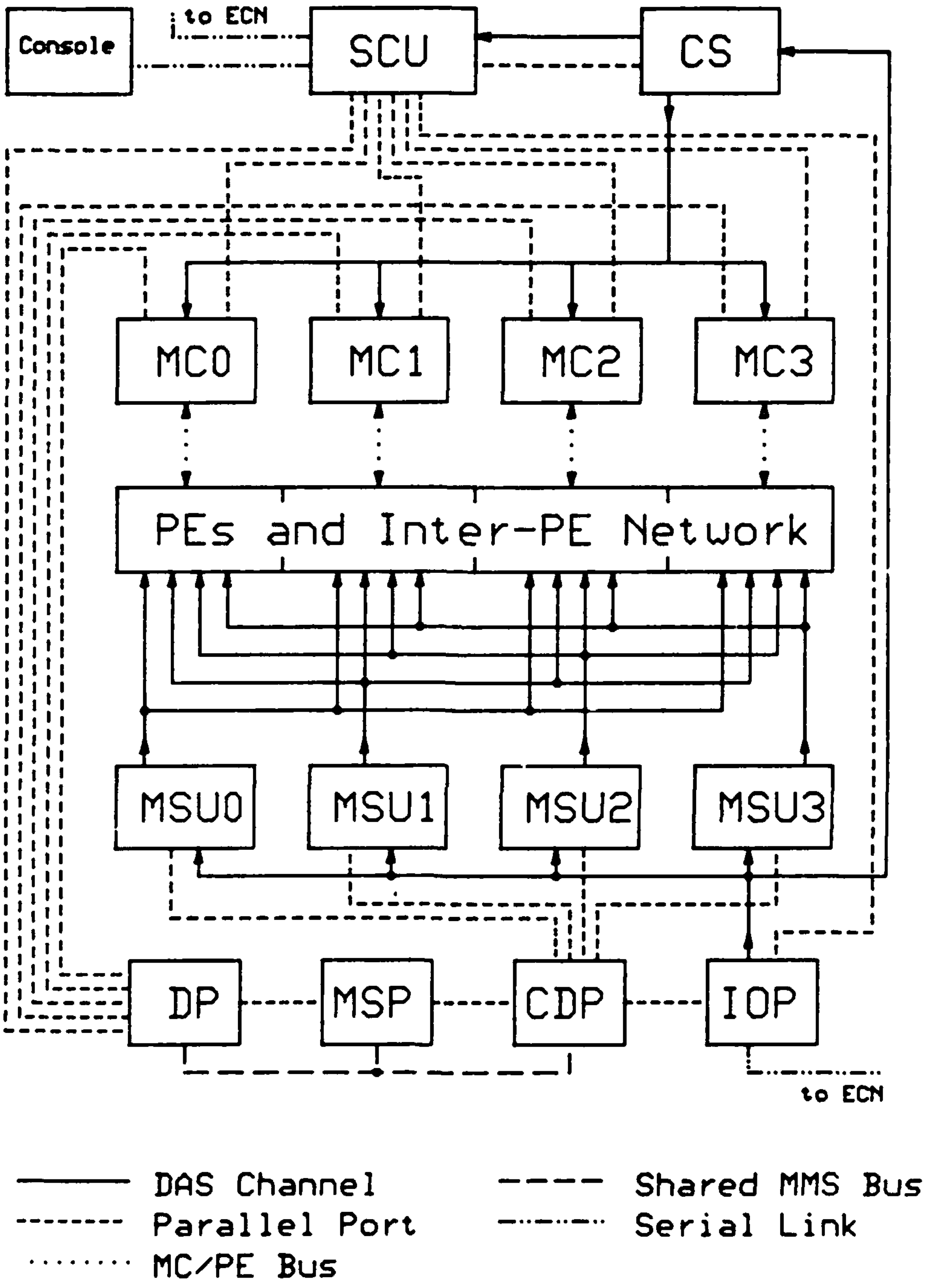


Figure 4-8. PASM prototype communication and data channels.

4.3 Prototype Board Descriptions

4.3.1 The Disk Access Switch (DAS) Board

General Description of the DAS Board

DAS Boards interface the MSUs, the CS, and the IOP to the second port of remote dynamic or static memories. Up to three memories can be connected to a single DAS Board. These memories belong to a single processor. Thus the MCs each have two dynamic memories and the Fetch Unit memory attached to a DAS Board that is located in the CS. All other DAS Boards are each connected to a dynamic memory pair.

A simplified block diagram of the DAS Board is shown in Figure 4-9. The board is accessed by a processor through the *VMEbus Interface* via the VMEbus. *REG1* and *REG2* can be set by the processor and perform an address translation of the four most significant address bits A19-23. REG1 determines at which address the remote memory appears for the processor. The VMEbus Interface detects an access to the board if A19-23 of the processor address is identical to the value in REG1. REG2 holds the base address of the remote memory to be accessed.

If the same data has to be broadcast to a set of memories each of which is connected to a different DAS Board, REG1 of the appropriate DAS Boards can be set to the same value, and data is sent to all of the remote memories connected to these boards at the same time. Note that the base addresses of the individual remote memories can differ since the values of REG2 can be set individually.

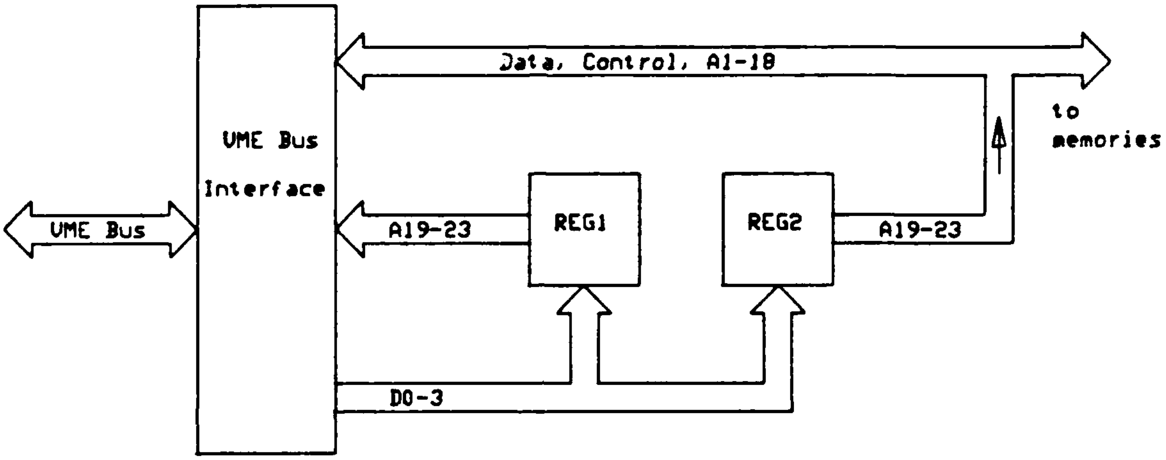


Figure 4-9. Simplified block diagram of the Disk Access Switch Board.

Design Tradeoffs

Since the only function of the DAS board is to provide a memory access channel from the CS, the IOP, or an MSU to the second port of a memory, few tradeoffs had to be made, and most of those involved implementation and not design. The current design could have been simplified if the address translation by the two register REG1 and REG2 would have been replaced by fixed address decoding. However, savings would have been small, and the broadcast capability of the DAS boards would have been lost. In the current implementation, each DAS board contains only a single channel, and the board is rather sparsely populated. In the early layout phase, it was therefore attempted to place two channels on a single board to reduce the board count. With the available two-sided printed circuit board capability, the layout complexity proved to be too high due to an excessive number of traces, and the one-channel-per-board approach had to be adopted.

Functional Description of the DAS Board

A detailed block diagram of the DAS Board is shown in Figure 4-10. It contains the following functional units: the *VMEbus Interface,* two write-only registers *REG1* and *REG2,* the *Comparator,* the *DAS Bus Interface,* and the *DTACK Combination Logic.* The operation of these components will now be explained in detail.

The board is accessed by the processor through the VMEbus Interface via the VMEbus. The VMEbus Interface buffers the data, address, and control lines of the VMEbus. It detects an access to the board if the data strobe VME.DS0* is asserted, the address modifiers VME.AM0-5 indicate an I/O

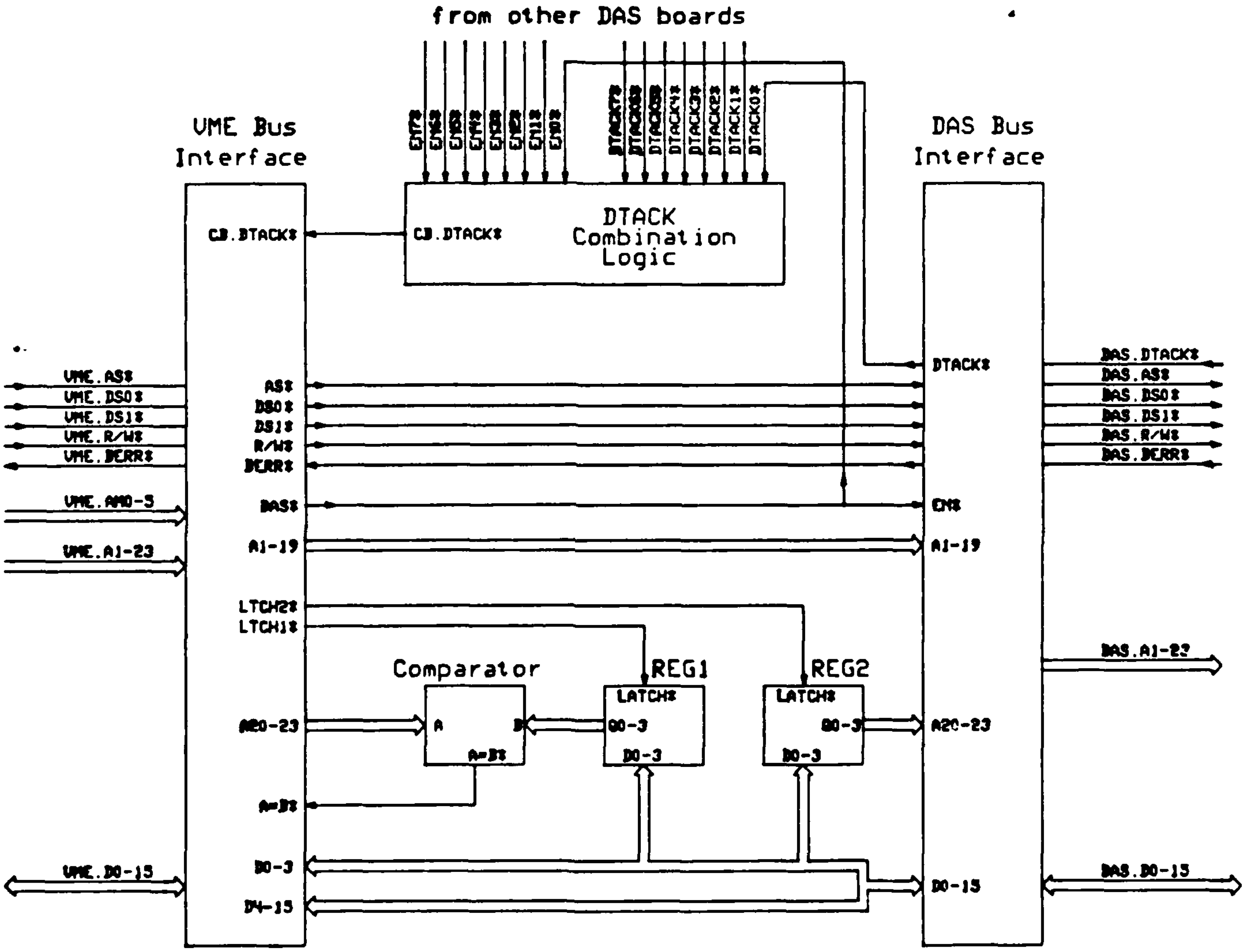


Figure 4-10. Detailed block diagram of the Disk Access Switch Board.

space access, and the address lines VME.A11-15 are identical to the board base address. The board address is settable by a four-bit switch. If the address lines VME.A8-10 have the value 000_2, the signal LTCH1* is asserted, thus latching the data bits D0-3 into REG1. If the address lines VME.A8-10 have the value 010_2, the signal LTCH2* is asserted, thus latching the data bits D0-3 into REG2. All other values for VME.A8-10 are reserved. If such an address is detected, the board will not respond, resulting in a timeout bus error. The data transfer acknowledge line VME.DTACK* is asserted 200ns after LTCH1* or LTCH2* are asserted, indicating the end of the access to the CPU. The negation of VME.DS0* by the processor causes the VMEbus Interface to negate VME.DTACK*, thus completing the access.

An access to the DAS Bus and thus to the remote memory is performed if the Comparator detects the same value on A20-23 and the output of REG1. It then asserts A=B*. This signal enables data drivers in the VMEbus Interface and the DAS Bus Interface, with the R/W* line determining the direction of the data transfer. It also enables the output Q0-3 of REG2, and Q0-3 are placed on the DAS Bus address lines DAS.A20-23. The control lines AS*, DS0*, DS1*, and R/W* are forwarded to the remote memory by the DAS Interface, where an access is performed if the value of REG2 coincides with the memory board base address. Upon successful completion of the access, the remote memory passes back a DAS.DTACK*. The DTACK* signal is combined with DTACK* signals of other DAS Boards that were also performing an access. This situation occurs if the same data is broadcast to more than one memory. The combination circuitry exists only on one of the DAS Boards of an MSU or of the CS. As soon as all boards that were enabled (i.e., had the EN* signal asserted) have asserted their DTACK* signal, the DAS Board that contains the combination circuit passes the DTACK* signal on to the VMEbus. The CPU will then negate the address and data strobes which causes the negation of the DTACK* lines, thus completing the access.

If any of the memory boards that were accessed detects an error, it asserts the bus error line DAS.BERR*. The signal is forwarded to the VMEbus and causes the CPU to abort the access by negating address and data lines. The remote memories then negate the BERR* line, completing the access.

4.3.2 The Dynamic Memory Board

General Description of the Memory Board

The Memory Board is a dual ported dynamic memory board with automatic hidden refresh, with a capacity of 1M bytes, organized as 512K 16-bit words. It implements the main memory of every processor of the PASM system.

Figure 4-11 shows a simplified block diagram of the Memory Board. Data is stored in the *Memory Array,* which also provides data integrity protection through byte-wise parity. Three sources can request access to the Memory Array: the *VME Port,* the *DAS Port,* and the *Refresh Logic.* The VME Port connects the processor to the memory via the VMEbus, the DAS Port connects the memory to a Direct Access Switch Board via a DAS Bus, and the Refresh Logic performs the refresh cycles required for proper dynamic memory operation. Any source indicates a request to the memory array by asserting a RQ* line. The *Access Arbiter* arbitrates and grants the requests. A request is granted by assertion of a GRT* line. If only a single access is pending, it will

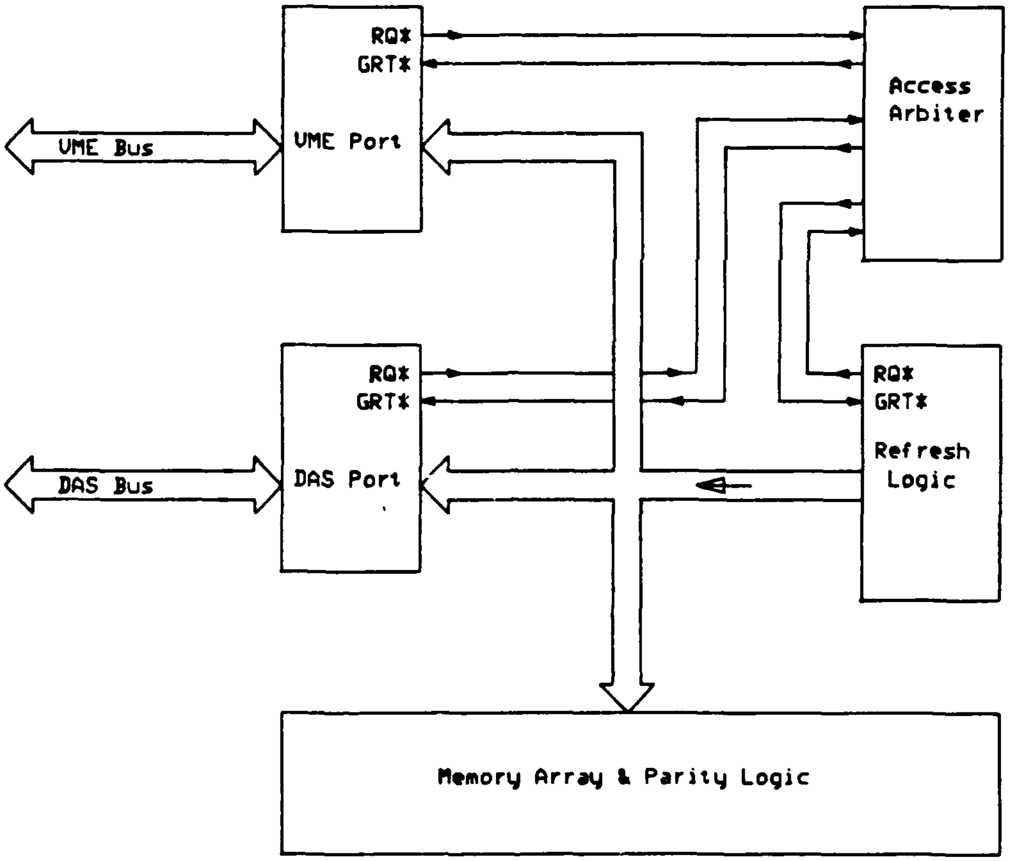


Figure 4-11. Simplified block diagram of the Dynamic Memory Board.

be granted immediately. If two or more requests arrive at the same time, the Access Arbiter will grant the requests sequentially. The refresh request takes priority over DAS Port and VME Port requests which have equal priority.

Design Tradeoffs

The Dynamic Memory Board is the main memory or all processors in the system. As such, it should provide as much memory as available. Due to the falling dynamic RAM prices, 256K bit RAM chips could be used, giving each memory a capacity of 1M byte. Double-buffering of the memory was desired to overlap computation and I/O, and thus a second port was provided on the board. Since the board adhères to the VMEbus standard, the VMXbus standard was considered for the second port. This standard defines all bus signals for a second memory port. It is, however, intended as a private CPU bus, and thus only physically short busses are permitted. In the PASM system, the interconnections between DAS board and memory are up to 1.5 m long, and thus the VMXbus was discarded. Instead, a bus was designed that uses differential line drivers for all time-critical signal lines. The design has proven to be reliable for all required cable lengths.

The CPU Boards that were purchased do not contain memory management. Therefore the original memory design (and the first memory prototype) contained memory mapping and protection hardware. Due to implementation constraints (space on the printed circuit board) this memory management had to be excluded from the final design. New developments like the inclusion of shared memory require memory management on the CPU Board so that the solution placing memory management on the Memory Boards would become insufficient. Ways to upgrade the CPU Boards to include memory management are currently studied.

One important consideration with dynamic memory boards is the kind of refresh being used. Since dynamic RAM chips store their information in MOS capacitors, the information is lost if it is not periodically refreshed. In an SIMD multiprocessor environment, the refresh has to be synchronized throughout the system. Otherwise, if each processor started a refresh cycle randomly, some processor would always be refreshing. Any refreshing processor would stop the other processors in its partition for the duration of the refresh. If many processors are in a partition, the machine would spend most of the time waiting for some processor to complete its refresh.

The 256K bit dynamic memory chips that are used in the prototype require 256 refresh cycles every 4 milliseconds. In each cycle, a row of 1024 memory bits is refreshed. Four ways can be used to accomplish the refresh:

(a) Software refresh through the CPU

(b) Refresh using a DMA controller

(c) Hardware burst refresh

(d) Hidden refresh

The software refresh is executed by the CPU and has the advantage that it requires no refresh hardware. This method was discarded due to the high overhead involved. To perform a refresh, the processors has to be interrupted every four milliseconds, and in the PASM system where each processor contains 2M byte of dynamic RAM, 1024 No-Operation instructions have to be executed during the interrupt routine. This performance degradation is not tolerable.

Refresh using a DMA controller can be done by performing DMA memory accesses to consecutive memory locations. Several problems can be identified.

The method relies on the existence of a DMA controller, and not all processors in the PASM system have one. In current implementation of the prototype, DMA controllers are not available at all. The DMA controller has to gain access of the CPU bus in order to perform the refresh access. This induces overhead, and is problematic in the PEs. If PEs execute an SIMD program, they can be disabled, and during that time they will not yield the system bus. Since PEs can be disabled for a long time, memory content could be lost. Thus this method was discarded also.

Hardware burst refresh is performed on a memory board without external support. Memory accesses to the board are halted every four milliseconds, and 256 refresh cycles are performed under hardware control. A common clock must be used by all PE memories so that refresh operations do not cause processing delays. The one disadvantage of this scheme is that memories become unavailable for the complete duration of the refresh. This might be problematic during high-speed I/O operations when data has to be accepted by PE memories. However, this method was considered for the prototype.

Hidden refresh attempts to perform the refresh without delaying the processor. Every 16 microseconds a refresh cycle is requested. If the CPU uses the memory, the refresh cycle is delayed until the CPU has completed the access. If the CPU is not using the memory, the refresh cycle is executed immediately. It is possible that the CPU attempts an access while a refresh cycle is in progress. In that case, the CPU is delayed, but never more than for the duration of a refresh cycle which is shorter than a regular memory access. There is a high probability that the processor is not delayed by the refresh since it performs not only accesses to the dynamic memory but also accesses I/O and spends time in instruction execution. The probability to hide th refresh is especially high for PEs in SIMD mode since they fetch all instructions from the SIMD instruction queue and only data from memory. If a refresh is performed during an SIMD queue access, it is hidden from the CPU. As with the hardware burst, refresh requests of the PEs have to be synchronized to avoid addition of refresh delay times. Since the hidden refresh will not block accesses for an extended time period and will often not influence the CPU at all, it was preferred over the burst refresh.

Functional Description of the Memory Board

Figure 4-12 shows a detailed block diagram of the Memory Board. Seven functional blocks can be identified: *Access Arbiter, Refresh Logic, CPU MUX, DAS MUX, Control Logic, Parity Logic,* and *Memory Array.* These blocks will now be described.

Access Arbiter. The memory can be accessed by three sources: the VMEbus, the DAS Bus, and the Refresh Logic. The Access Arbiter decides which of a pending access is served next. To ensure data integrity, the refresh takes priority over the other two sources, and VMEbus and DAS Bus have equal priority. A VMEbus access is pending if the address strobe VME.AS* is asserted, the address modifiers VME.AM0-5 indicate a standard memory access, and the address lines VME.A19-23 are identical to the VME side board address. This address is encoded in a PAL. A DAS Bus access is pending if the address strobe DAS.AS* is asserted and the address lines DAS.A19-23 are identical to the DAS side board address. The DAS side board address can be set by four jumpers on the board. DAS side board address and VME side board address can be different. A refresh access is pending if the refresh request signal

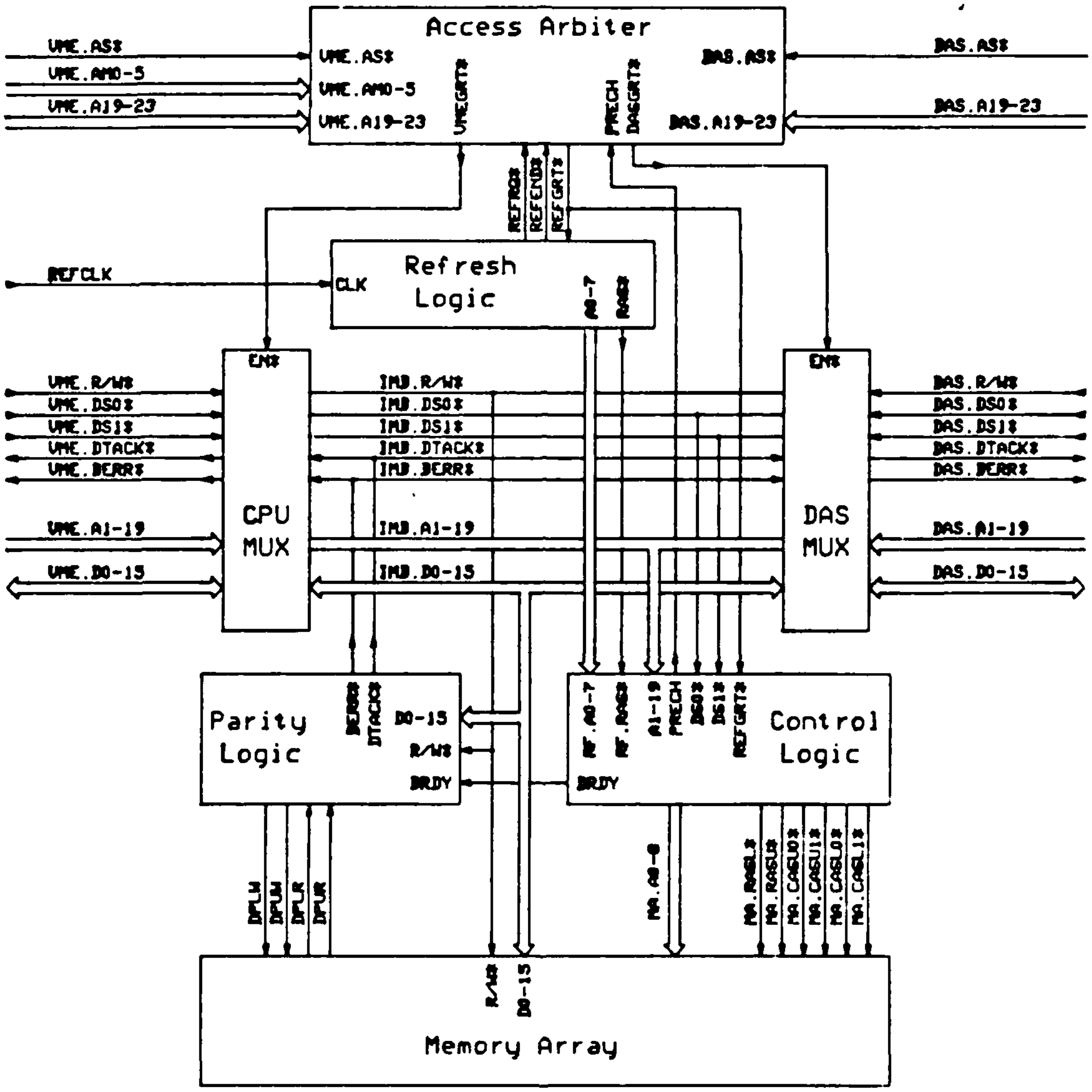


Figure 4-12. Detailed block diagram of the Dynamic Memory Board.

(REFRQ*) is asserted by the Refresh Logic.

As soon as one or more pending accesses are detected, the Access Arbiter will grant an access to a source by asserting one of the grant signals VMEGRT*, DASGRT*, or REFGRT*. A grant signal enables the VME MUX, DAS MUX, or Refresh Logic. An access from VME or DAS side ends when the appropriate address strobe is negated. Further accesses are prohibited until the precharge line (PRECH) line is negated. This signal is required due to dynamic RAM timing constraints. The end of the refresh cycle is signaled by the assertion of the REFEND* signal. The precharge time requirement of a refresh request is fulfilled by the Refresh Logic.

Refresh Logic. The memory requires 256 refresh cycles every four milliseconds. This is accomplished by performing one refresh cycle every 16 microseconds. A transition of the externally generated clock signal REFCLK starts a refresh cycle, and the REFRQ* signal is asserted. As soon as the grant REFGRT* is received, the contents of an eight-bit counter are placed onto the address bus RF.A0-7, and the RF.RAS* signal is asserted. After the refresh cycle time has expired, RF.RAS* is negated, the counter is incremented by one, and REFEND* is asserted and REFRQ* is negated after the precharge time has been satisfied. As soon as REFGRT* is negated, REFEND* is negated, and the refresh cycle is completed.

VME MUX. When the Access Arbiter grants access to the VMEbus side, the VMEGRT* signal enables the VME MUX, and control, address, and data signals are placed on the *Internal Memory Bus (IMB)*.

DAS MUX. When the Access Arbiter grants access to the DAS Bus side, the DASGRT* signal enables the DAS MUX, and control, address, and data signals are placed on the Internal Memory Bus.

Control Logic. The Control Logic generates all timing signals required by the dynamic memory array. As soon as one or both of the data strobes IMB.DS0* or IMB.DS1* are asserted, a memory cycle is started. Address bits IMB.A0-A8 are placed onto the memory address bus lines (MA.A0-8), and one or both rcw address strobes (MA.RASL*, MA.RASU*) are asserted, depending on the state of the data strobes. Following this, addresses A9-18 are placed on the MA.A0-8 lines, and the proper Column Address Strobe (MA.CASL0*, MA.CASL1*, MA.CASU0*, MA.CASU1*) is asserted. The state of IMB.DS0*, IMB.DS1*, and IMB.A19 decide which of the four CAS* signals are asserted. After the memory access time has expired, the DRDY signal is asserted. The Control Logic stays in this state until the external source negates the data strobes. Then RAS*, CAS*, and DRDY signals are negated immediately. The PRECH signal is negated after the precharge time has expired, and the memory cycle is completed.

During a memory refresh, the Control Logic is bypassed. The refresh addresses RF.A0-7 are placed on the memory address lines MA.A0-7, and the RF.RAS* signal is placed on both MA.RASL* and MA.RASU*. The Refresh Logic provides the proper timing.

Parity Logic. The Parity Logic provides single error detection through the use of byte-wide parity generation/checking. One parity bit is associated with the low and one with the high data byte. This facilitates correct operation if single bytes and not complete words are written.

During a write cycle, parity bits (DPLW, DPUW) are derived from the data bits and are written into the memory together with the data bits. As soon as the DRDY signal is asserted, the parity logic asserts the data transfer acknowledge (DTACK*) signal, informing the external source that the memory cycle has ended.

During a read cycle, the data bits and parity bits (DPLR, DPUR) are read from the memory, and the stored parity bits are compared with parity generated from the data bits that were read. The data strobes are needed to determine which of the data bits was read and thus has to be checked. As soon as the DRDY signal is asserted, the DTACK* signal is asserted if the parity was correct, or the bus error (BERR*) signal is asserted if a parity error was detected. BERR* and DTACK* both cause the external source to terminate the access. As soon as a negated DRDY is detected, BERR* or DTACK* are negated.

Memory Array. The memory array consists of 36 256K bit dynamic memory chips, 32 for storing data, and four for holding parity. It is organized as two banks of 18 bits each (low byte and parity, high byte and parity). The Control Logic provides all signals required for proper operation.

4.3.3 The Fetch Unit

General Description of the Fetch Unit

The Fetch Unit is a board in each MC that is used in SIMD mode to broadcast instructions from the MC to its associated PEs. A simplified block diagram is shown in Figure 4-13. The MC CPU can access the *Fetch Unit Controller*, the *Mask Register* and the *Queue Unit* via the *VMEbus Interface*. The processor can instruct the Fetch Unit Controller to send a block of instructions from the Fetch Unit Memory to the Queue Unit. The block is specified by a start address and the number of words to be sent. The operation is

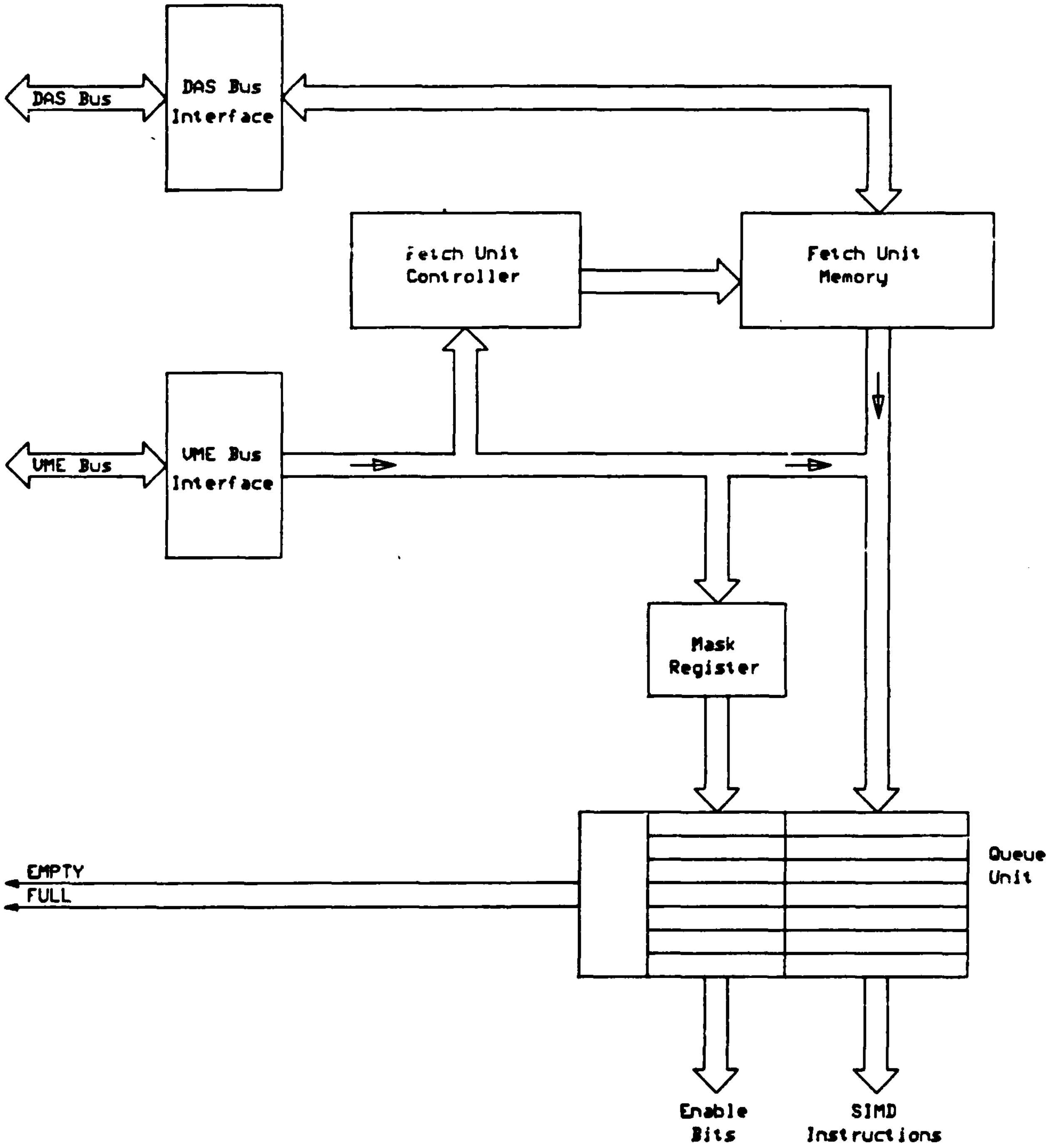


Figure 4-13. Simplified block diagram of the Fetch Unit

performed automatically without CPU intervention. Whenever an instruction word is enqueued in the Queue Unit, the current content of the Mask Register is enqueued with it, so that each SIMD instruction has an enable vector associated with it. Thus, the processor has to load the Mask Register every time a new enable vector is required. Sometimes it is necessary for the MC processor to enqueue instructions directly instead of instructing the Fetch Unit Controller to move them from memory into queue. For these cases, a direct path from the processor into the Queue Unit is provided. The status of the Queue Unit (empty/full) can be read by the processor. The Fetch Unit Memory is loaded by the Control Storage via the DAS Bus and the DAS Bus Interface.

Design Tradeoffs

The Fetch Unit is by far the most complex of all boards designed for the prototype. Since only four Fetch Units are needed they were implemented in wire-wrap technology. Even though wrapping of such a board is a tedious task, it has proven to be a good decision since many changes had to be made from the original design, and the wire-wrap board facilitates these changes. The basic tradeoffs that lead to the current Fetch Unit structure have been described in [Kue86]; still a number of tradeoffs had to be made in order to implement the board.

Finite State Machine Control. The original design of the Fetch Unit anticipated the use of a fast bit-slice microprocessor as the Fetch Unit Controller. This would have required a large number of chips and the development of microprograms which are difficult to debug. Analysis showed that a finite state machine implemented by PALs could provide the same functionality at much lower complexity and cost.

Fetch Unit Queue. The necessary length of the FIFO Queue was determined by simulations [KuS82], and 64 words are sufficient. Originally, the queue was designed using dual-ported ECL RAMs and counters that point to locations into the RAMs. This design was very complex and would have been difficult to implement. Fortunately, integrated FIFO chips became available that are 4 bits wide and 64 words deep. Thus the FIFO design could be reduced to six FIFO chips (four for the SIMD instructions, eight for the enable bits) and some glue logic.

Memory. The Fetch Unit contains static RAM that holds SIMD instructions which are broadcast to the PEs in SIMD mode. The original design called for 128K bytes of memory, but space constraints on the wire-wrap board forced a reduction to 32K byte. All SIMD algorithms that have been studied by the PASM group so far need only a small portion of this reduced memory area so that all these algorithms can be run on the machine. If bigger SIMD programs should be devised, an overlay technique can be used. Only part of the SIMD program would reside in memory, and the CS would be used to swap out unneeded parts and swap in necessary program segments. Since only four Fetch Units are needed, a redesign would be possible after the current implementation has been proven to work reliably.

The memory is not physically double buffered. In contrast to all dynamic memories in the system, the Fetch Unit does not contain two separate memory modules one of which can be used for I/O while the other is used to broadcast instructions. Logically, the memory is doubled buffered since both Fetch Unit and the Control Storage can access the memory simultaneously, and the access is granted in an alternating fashion. This increases the average time to enqueue an SIMD instruction into the queue while I/O is done, but due to the fast memories used for the Fetch Unit memory this will not create a bottleneck, i.e., the Fetch Unit will fill the queue faster than the PEs can drain it. The main reasons for using only logical double buffering are the use of fast memories and space limitation of the wire-wrap board. If more memory (and possibly slower memory) is used, physical double buffering has to be reconsidered.

Functional Description of the Fetch Unit

A detailed block diagram of the Fetch Unit is shown in Figure 4-14. The Fetch Unit contains the following functional units: *VMEbus Interface; Fetch Unit Controller; DAS Bus Interface; Count Register; 8-bit Adder; Base Register; Address Register; Memory Array; Memory Data Register; CPU Data Register; Mask Register; Queue Unit* composed of *FIFO Control, Mask FIFO,* and *Instruction FIFO;* and *Queue Output Control.* The function of these components and their interactions will now be described in detail.

The VMEbus Interface connects the VMEbus and thus the CPU to the Fetch Unit. The VMEbus Interface buffers the data, address, and control lines of the VMEbus. It detects an access to the board if the address modifiers VME.AM0-5 indicate an I/O space access, the address strobe VME.AS* and both data strobes VME.DS0* and VME.DS1* are asserted, the VME.R/W* line indicates a read access, and the address bits VME.A11-15 are identical to the board base address which is encoded in a PAL. The board base address is $FF0000_{16}$. The address lines A1-10 determine which of the registers on the Fetch Unit is accessed. Since both data strobes have to be asserted during an

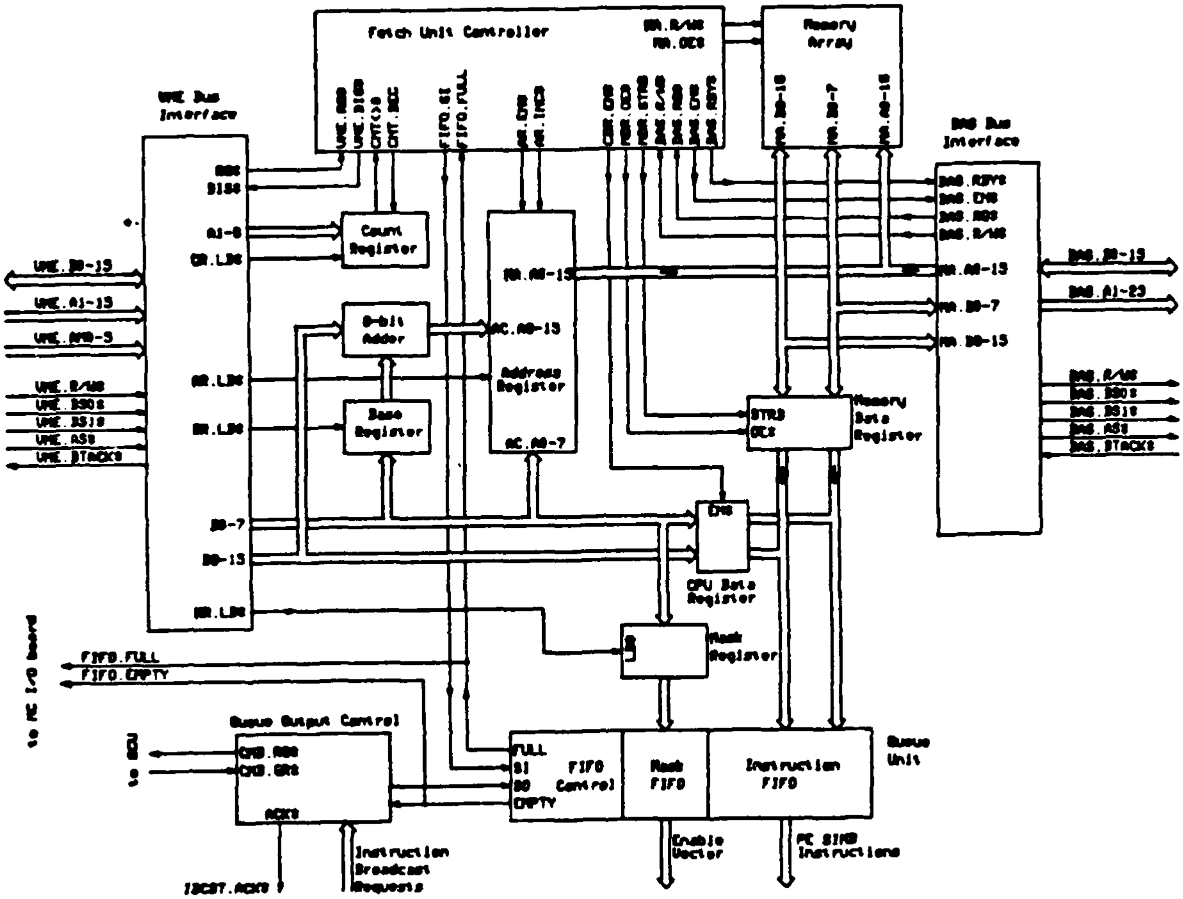


Figure 4-14. Detailed block diagram of the Fetch Unit.

access, only 16 bit accesses are possible even if only eight bits are actually used. All registers on the Fetch Unit are write-only. If a register is selected, its load line is asserted. The load lines are CR.LD* (Count Register), AC.LD* (Address Register), BR.LD* (Base Register), MR.LD* (Mask Register), or CPU.RQ (access to the queue through the CPU Data Register).

Accesses to the Count Register and to the Address Register are handled in a special way. If a location in the range from $FF0000_{16}$ to $FF01FE_{16}$ is accessed, the lower eight address bits VME.A1-8 are strobed into the Count Register; VME.D0-7 is strobed into the lower eight bits of the Address Register; VME.D8-15 is added to the contents of the Base Register and, the result is strobed into the upper eight bits of the Address Register. Thus Address Register and Count Register are loaded in a single memory cycle, and addressing of the count register is implicit.

After the data has been latched into the appropriate register, the VMEbus Interface asserts the data transfer acknowledge line VME.DTACK*, indicating that the transfer can be terminated. When AS*, DS0* and DS1* are negated by the CPU, the access is completed and the VMEbus Interface negates VME.DTACK*. In the following, "loading " of a register refers to the above sequence of events.

In order to move SIMD instructions and their enable vectors from the Memory Array into the Queue Unit, and from there to the PEs, the following sequence of actions has to be performed by the CPU. The Base Register is loaded with an eight bit address offset, and the Mask Register is loaded with the enable vector. Then the Address Register and the Count Register are loaded simultaneously. Whenever the Count Register contains a non-zero value, the signal CNT<>0 is asserted. This informs the Fetch Unit

Controller (FUC) that SIMD instructions have to be transferred from the Memory Array into the Queue Unit. The FUC inhibits further accesses to the Fetch Unit by asserting VME.DIS*. If the CPU performs an access to the Fetch Unit while VME.DIS* is asserted, the access will be delayed until VME.DIS* has been negated.

Every time a new data item is to be moved from the Memory Array to the Queue Unit, the FUC asserts AC.EN*, placing the current content of the Address Register onto the memory address lines MA.A0-15. It also asserts the memory output enable line MA.OE* and sets the memory read/write line MA.R/W* to "read." After waiting for the memory access time, the FUC pulses the memory data register strobe MDR.STRB*, thus latching the data that was read from memory into the Memory Data Register. AR.EN* and MA.OE* are then negated. The FUC pulses the Address Register increment line AR.INC, incrementing the current Address Register content by one, and it pulses the Count Register decrement line CNT.DEC, decrementing the Count Register by one. The FUC then monitors the Queue Unit full indicator FIFO.FULL. As soon as the FUC detects that the FIFO is not full, it strobes the content of the Memory Data Register into the Instruction FIFO and the content of the Mask Register into the Mask FIFO by asserting MDR.OE* and then pulsing the FIFO shift-in signal FIFO.SI. After MDR.OE* has been negated, the FUC checks CNT<>0. If the signal is still asserted, another data transfer takes place. If the signal is negated, the FUC returns to its idle state by negating VME.DIS*.

If the FUC is not busy, the CPU can move data items directly into the Queue Unit. This is accomplished by loading the CPU Data Register. An access to this register asserts the request line CPU.RQ*, the FUC asserts

VME.DIS*, and it strobes VME.D0-15 into the Instruction FIFO by asserting the CPU Data Register enable line CDR.EN* and pulsing FIFO.SI as soon as a negated FIFO.FULL is detected. After the data has been strobed into the FIFO, the FUC negates VME.DIS* which is causes the assertion of VME.DTACK*, thereby terminating the access.

The CPU cannot access the Memory Array directly. Loading of the Memory Array is accomplished through a DAS channel. The DAS Bus Interface is connected to a DAS Board. An access to the Memory Array is performed if the address strobe DAS.AS* and one or both of the data strobes DAS.DS0* and DAS.DS1* are asserted, and if the address bits DAS.A19-23 coincide with the Fetch Unit DAS side board address which is decoded in a PAL. Then the memory access request DAS.RQ* is asserted, and the required type of access is indicated by the DAS.R/W* line. If the FUC does not currently use the memory, it grants access to the DAS Bus Interface by asserting DAS.EN*. The DAS Bus Interface then places data and address lines onto the busses MA.D0-7, MA.D8-15, and MA.A0-15. The FUC controls the access by setting MA.R/W* and MA.OE* properly, and by asserting DAS.RDY* when the access is complete. DAS.RDY* causes the DAS Interface to assert DAS.DTACK*, signaling the end of the access. When the DAS Bus data and address strobes are negated, the DAS Bus Interface negates its request, the FUC negates DAS.EN* and DAS.RDY*, and the Interface negates DAS.DTACK*.

Instructions and enable vectors that have been written into the Queue Unit are read by the PEs. The simultaneous execution of instructions in SIMD mode must be ensured, and the Queue Output Control handles this task. When all PEs of an MC group have asserted their instruction broadcast request, and the FIFO is not empty, the Queue Output Control asserts the CMB.RQ* line. This signal is forwarded to the Partition Combination Logic on the SCU I/O board. The Partition Combination Logic asserts the CMB.GRT* after all MC groups in a partition have asserted their CMB.RQ* signal. Thus synchronization of PEs across MC groups is ensured.

When the Queue Output Control receives the CMB.GR* signal, it asserts the Instruction Broadcast Acknowledge line IBCST.ACK*. When all instruction broadcast request lines have been negated (i.e., all PEs have accepted the instruction), the Queue Output Control negates CMB.RQ*. When receiving a negated CMB.GR*, the Queue Output Control negates IBCST.ACK* and makes the next item in the FIFO available by pulsing the shift-out signal FIFO.SO.

In some cases, the status of the Queue Unit must be known by the CPU. The EMPTY and FULL bits are therefore forwarded to the MC I/O board where they can be read by the CPU.

4.3.4 The SCU I/O Board

General Description of the SCU I/O Board

The SCU I/O Board performs several functions required for proper overall system operation. It generates a dynamic memory refresh clock that is distributed throughout the system so that all refresh operations in the system are performed simultaneously, reducing refresh overhead. It contains the *Partition Combination Logic* that combines SIMD instruction broadcast requests and condition code bits according to the current partitions. The board generates

RESET signals that selectively reset parts of the system under the control of the SCU, and it contains eight general purpose LEDs. Due to its straightforward design, no design tradeoffs will be discussed for the SCU I/O Board.

Functional Description of the SCU I/O Board

The SCU I/O Board is shown in Figure 4-15. It contains the following functional units: *VMEbus Interface; Parallel Port; Memory Refresh Clock Generator; Partition Combination Logic* consisting of the *Instruction Broadcast Combination (IBC) Logic, Condition Code Combination (CC) Logic,* and the *Condition Code Synchronization (CCS) Logic;* eight general purpose LEDs, and the *System RESET Distribution.*

The VMEbus Interface buffers the data, address, and control lines of the VMEbus. It detects an access to the Parallel Port if the address strobe VME.AS* and the data strobe VME.DS0* are asserted, the address modifiers VME.AM0-5 indicate an I/O space access, and VME.A11-15 coincide with the SCU I/O Board address. The board address is set to $FF4000_{16}$ by four switches on the board. The The interface then asserts the chip select line CS*, and data is transferred to or from the Parallel Port depending on the value of the VME.R/W* line. The address lines VME.A1-5 are placed on the Parallel Port address lines A0-4, selecting one the the chip's 32 internal registers. After the Parallel Port has completed the access, it asserts DTACK* which is forwarded to the CPU as VME.DTACK*. The CPU negates its address and data strobes, causing negation :f CS* and DTACK*, and the access is completed.

Port B of the Parallel Port is used as an output, and bits PB0-3 determine the current machine partitioning. According to PB0-3, the Instruction

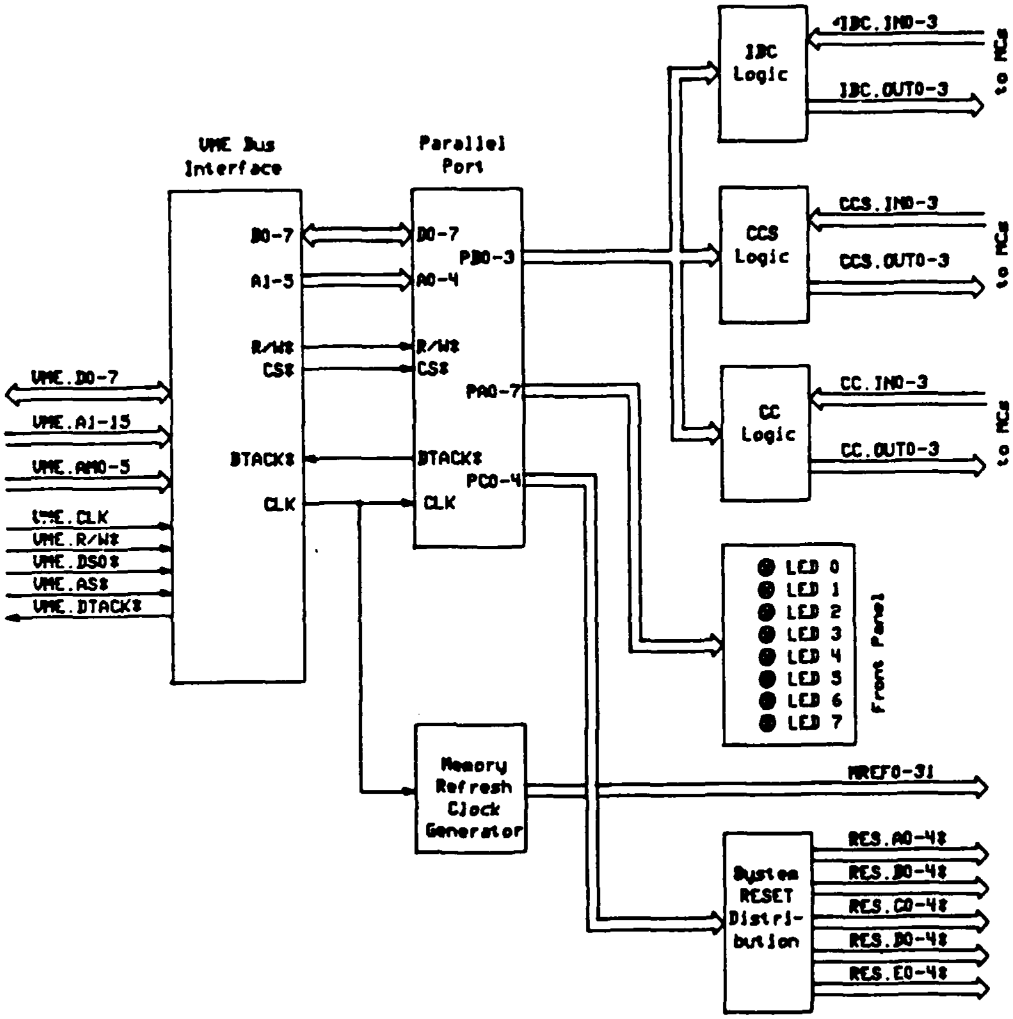


Figure 4-15. Block diagram of the SCU I/O Board.

Broadcast Combination signals IBC.IN0-3, the Condition Code Synchronization signals CCS.IN0-3, and Condition Code signals CC.IN0-3 are combined and the combined signals are placed on IBC.OUT0-3, CCS.OUT0-3, and CC.OUT0-3, respectively.

The eight LEDs can be set by using Port A of the Parallel Port as an output and setting bits PA0-7 as desired. An output of "one" will result in a dark, and output of "zero" will result in a lit LED.

Bits PC0-4 are used to selectively reset parts of PASM. Five reset busses with five lines each are available. If PC0 is logical "zero," the RES.A0-4* lines are asserted, resetting all connected devices. PC1-4 work similarly for the other four reset busses.

The memory refresh clock generator provides 32 clock signals with a cycle time of 16 microseconds. This clock causes simultaneous dynamic memory refreshs everywhere in the system.

4.3.5 The Network Interface Board

General Description of the Network Interface Board

A simplified block diagram of the Network Interface Board is shown in Figure 4-16. The processor accesses the board through the VMEbus via the *VMEbus Interface,* sends data to the network through the *Input Stage Port (ISP),* and receives data from the network via the *Output Stage Port (OSP).*

To send a data item to the network, the CPU writes a 16-bit word to the *ISP Parallel Port.* The *Parity Generator* adds two parity bits to the word, and data and parity are sent to the network through the *Input* Stage or the *Input*

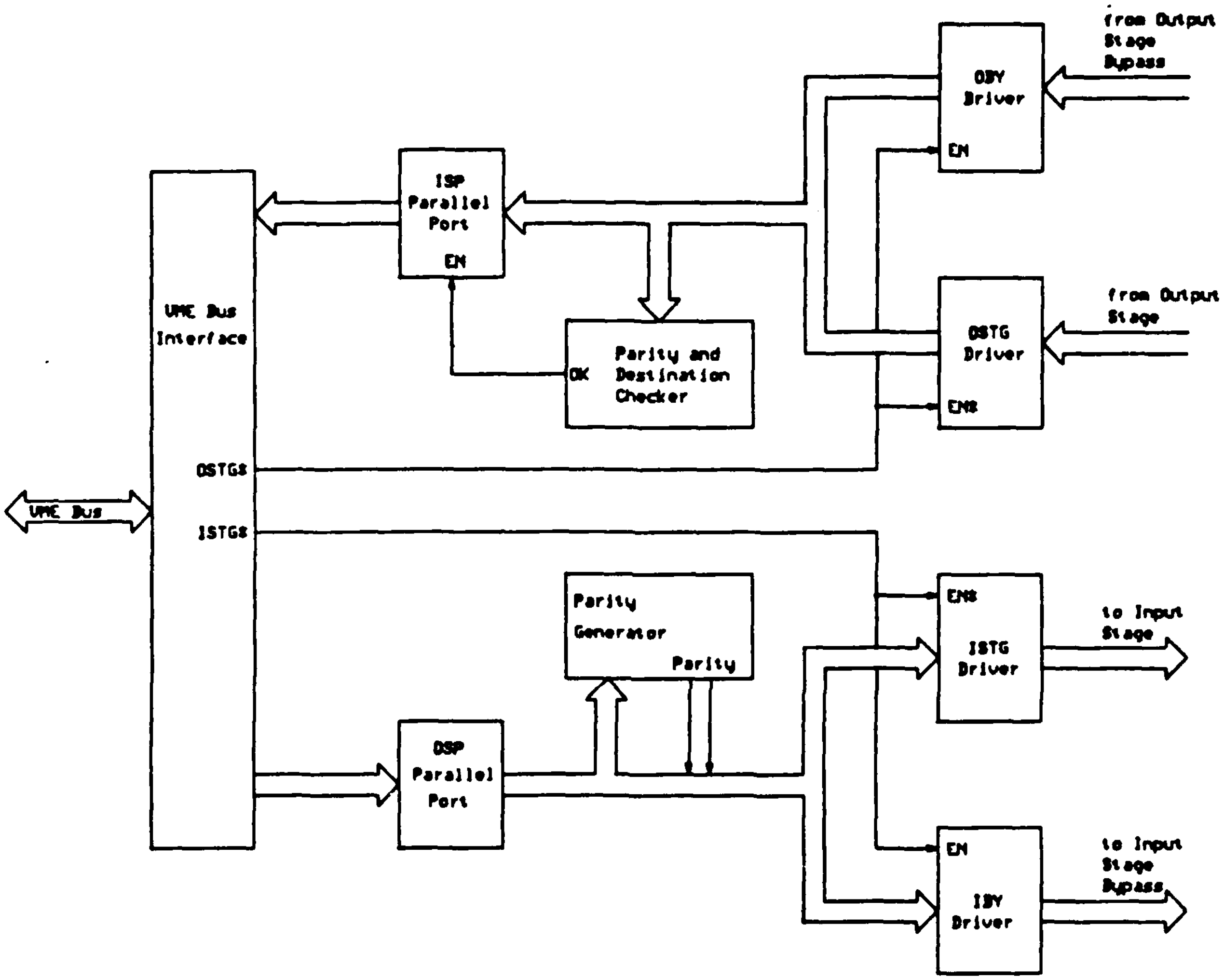


Figure 4-16. Simplified block diagram of the Network Interface Board.

Stage (ISTG) Driver, depending on the status of the ISTG* line. If ISTG* is asserted, the input stage of the network is enabled, and the ISTG Driver is used. If the ISTG* signal is negated, the network input stage is disabled, and the IBY Driver is used.

Data from the network reaches the Output Stage Port through the *Output Stage Bypass (OBY) Driver* or the *Output Stage (OSTG) Driver,* depending on the state of the OSTG* line. If OSTG* is asserted, the output stage of the network is enabled, and the OSTG Driver is used. If the OSTG* signal is negated, the network output stage is disabled, and the OBY Driver is used. The *Parity and Destination Checker* asserts the signal OK if if the message was destined for this port, and the parity is correct. The asserted OK signal enables the *OSP Parallel Port,* and the CPU can read the data via the VMEbus Interface.

Design Tradeoffs

The design of the VMEbus Interface and the layout of this board was supervised by this author, while the actual network interface was designed under the supervision of Nat Davis. Design tradeoffs concerning the network interface can be found in [Dav85].

Functional Description of the Network Interface Board

A detailed block diagram of the Network Interface Board is shown in Figure 4-17. The *VMEbus Interface* connects the VMEbus to the *Output Stage Port* and the *Input Stage Port* as described in the general description of the Network Interface Board. The ISP Parallel Port and the OSP Parallel Port as

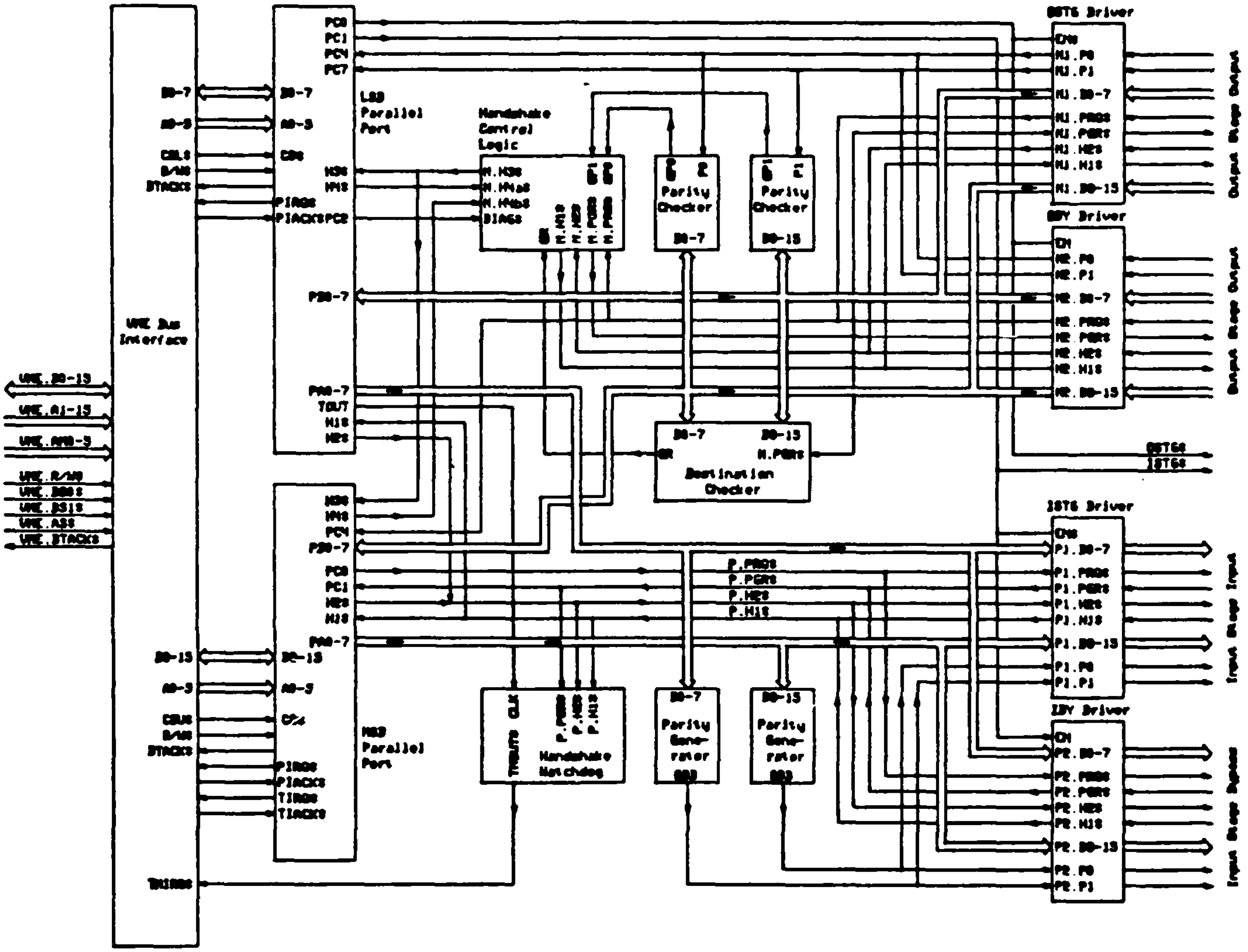


Figure 4-17. Detailed block diagram of the Network Interface Board.

introduced in the general description are implemented by two MC68230 parallel port chips, the *LSB Parallel Port* and the *MSB Parallel Port.* The LSB Parallel Port is connected to the lower eight data bits (D0-7), while the MSB Parallel Port is connected to the upper eight data bits (D8-15). The I/O ports PA of both chips are associated with the Input Stage Port, while the I/O ports PB are associated with the Output Stage Port.

The Input Stage Port consists of the I/O ports PB of the LSB and MSB Parallel Ports, the *Handshake Watchdog,* two *Parity Generators,* the *Input Stage (ISTG) Driver,* and the *Input Stage Bypass (IBY) Driver.* The Output Stage Port consists of the I/O ports PA of the LSB and MSB Parallel Ports, the *Output Stage (OSTG) Driver,* the *Output Stage Bypass (OBY) Driver,* two *Parity Checkers,* the *Destination Checker,* and the *Handshake Control Logic.*

The Input Stage Port is connected to the first stage of the network, the Output Stage Port is connected to the last stage. From a logical standpoint, the Input Stage Port is connected directly to the Output Stage Port, since the network does not change but only delay signals fed through it.

The VMEbus Interface buffers the data, address, and control lines of the VMEbus. It detects an access to the board if the address modifiers VME.AM0-5 indicate an I/O space access, the address strobe VME.AS* is asserted, and the address lines VME.A11-15 coincide with the board address $FFA000_{16}$ which is encoded into a PAL on the board. If the data strobe VME.DS0* is asserted, the LSB Parallel Port is selected by assertion of the signal CSL*. If VME.DS1* is asserted, the MSB Parallel Port is selected by assertion of the signal CSU*. The direction of the data transfer is controlled by the read/write line R/W*, and the selection of the internal register is determined by the address lines VME.A1-5. When a parallel port has placed data on the data bus duriug a read cycle, or read data from the data bus during a write cycle, it asserts its data transfer acknowledge line DTACK*. Both DTACK* signals are combined and forwarded to the processor as the VME.DTACK* signal. When receiving VME.DTACK*, the CPU terminates the access by negating data and address strobes. The parallel ports in turn negate their DTACK*, thus completing the access.

The LSB Parallel Port can generate a vectored interrupt on level 4 by asserting PIRQ*. The MSB port has the PIRQ* interrupt on level 4 as well, and in addition has a timer interrupt on level 3 (TIRQ*). The VMEbus Interface asserts the appropriate interrupt request lines VME.IRQ3* and VME.IRQ4* when the ports issue an interrupt request. The CPU responds to these interrupts by placing the interrupt level on VME.A1-3 and asserting VME.IACKIN*. The VMEbus Interface then asserts the appropriate PIACK* or TIACK*, which causes the interrupt vector to be placed on the data bus. Details of the interrupt handling of the parallel ports are described in the MC68230 manual. The handshake watchdog generates a third interrupt, TMIRQ*. This interrupt is forwarded to the PE I/O board where it is processed further.

To send a message through the network, a path must be established since the network is circuit switched. First, a routing tag is placed in PA0-7 of the MSB Parallel Port and a broadcast tag is placed in PA0-7 of the LSB Parallel Port. The port then automatically asserts the handshake signal H2*, indicating that data is available. A parity bit is appended to the lower and the upper eight data bits by the two Parity Generators, and the handshake signal, data, and parity are forwarded to the network through the ISTG Driver or the IBY Driver. The selection of the path is accomplished through PC1 = ISTG* of the

LSB ports. If ISTG* is asserted, the input stage of the network (and thus the ISTG Driver) is used. If ISTG* is negated, the input stage is bypassed and the IBY Driver is used.

After routing and broadcast tags have been written to the parallel ports, the path request line PRQ* is asserted by placing a '0' on PC0 of the MSB port. PRQ* is passed through the driver and establishes the path if possible. The message reaches an Output Stage Port after traversing the network. It enters the Output Stage Port through either the OSTG Driver if the output stage is enabled, or otherwise through the OBY Driver. If the signal PC0 = OSTG* of the LSB port is '0', the output stage is enabled, otherwise it is disabled. OSTG* enables the proper driver.

As soon as the Output Stage Port detects the assertion of the PRQ* signal, it asserts the path grant signal PGR* which is passed back to the source. Then the parity is checked by two Parity Checkers. If the parity is correct, the checkers assert the good parity signals GP0 and GP1. The routing and broadcast tags are analyzed by the Destination Checker. If the message was destined for the port it arrived at, the good routing signal GR is asserted and latched in the Handshake Control Logic. While PRQ* remains asserted, GR does not change. If GR, GP1, GP1, and N.H2* are asserted, the Handshake Control Logic asserts the signal N.H3*, indicating to the MSB and LSB Parallel Ports that data is available. The parallel ports latch the data and negate H4*, indicating they have received the data. The Handshake Control Logic then asserts N.H1*, which is passed back to the Input Stage Port in conjunction with PGR*.

At the Input Stage Port, the path grant line PGR* is monitored at PC1 of the MSB port. As soon as the signal is asserted, the Input Stage Port knows a path has been established and can start regular data transfers. Also at the Input Stage Port, the Handshake Watchdog starts a timer as soon as it detects that P.PGR* and P.H1* are asserted (i.e., a path is established and data is being transferred). The timer is clocked by the square-wave output TOUT of the LSB Parallel Port, and thus the timeout duration is determined by software. The timer is stopped as soon as P.H1* is asserted, indicating the transfer is completed. If no P.H1* is received before the watchdog times out, an interrupt TMOUT* is generated, and the CPU has to take proper action (e.g., retry). If the Handshake Watchdog times out during the path setup, the message has reached an incorrect destination or had erroneous parity.

After a path has been established, regular data transfers can take place along the path established by the above procedure. They are controlled by the handshake line H1*-H4* in the same fashion as the transfer of routing and broadcast tag. The Output Stage Port accepts data only if GP0 and GP1 are asserted. If a parity error occurred during the transfer, no H1* signal is passed back to the Input Stage Port, and the Handshake Watchdog will time out.

After all desired data items have been transferred, the MSB Parallel Port of the Input Stage Port negates PRQ*, and the network drops the path. If required, the Output Stage Port can detect the negated PRQ* signal by polling PC4 of the MSB parallel port.

Fault tolerance is an important issue in the interconnection network as was already indicated by the two independent paths to the network. If a fault has been detected in the network (e.g., a message arrived at an incorrect destination or the message parity is incorrect even after a few retries), the fault must be located so that it can be bypassed. For a detailed discussion of fault location techniques, see [DaH85]. The Network Interface Board provides hardware support for fault location. Port C bit PC2 of the LSB Parallel Port can assert a diagnosis line DIAG* in the Handshake Control Logic. If this line is asserted, data is strobed into the Output Stage Port even if the parity or the routing are incorrect. The Output Stage Port parity bits can be monitored at PC4 and PC7 of the LSB Parallel Port, and the PRQ* signal of the Output Stage Port is available at PC4 of the MSB Parallel Port.

4.3.6 The Parallel Port Board

General Description of the Parallel Port Board

The Parallel Port Board provides communication channels between various processors in the system. These channels are used exclusively by operating system routines to exchange control information.

Figure 4-18 shows a simplified block diagram of the board. The processor accesses the board through the VMEbus via the VMEbus Interface. Four unidirectional ports, Port 1A, Port 1B, Port 2A, and Port 2B, are available, with the data transfer direction of any port determined in software. In the diagram, 1A and 2A are shown as output, 1B and 2B as input. This can be adapted to system needs. However, ports 1A and 1B and ports 2A and 2B are grouped together on two flat cables, so that 1A, 1B and 2A, 2B will usually have opposite data directions to facilitate bidirectional transfers between boards.

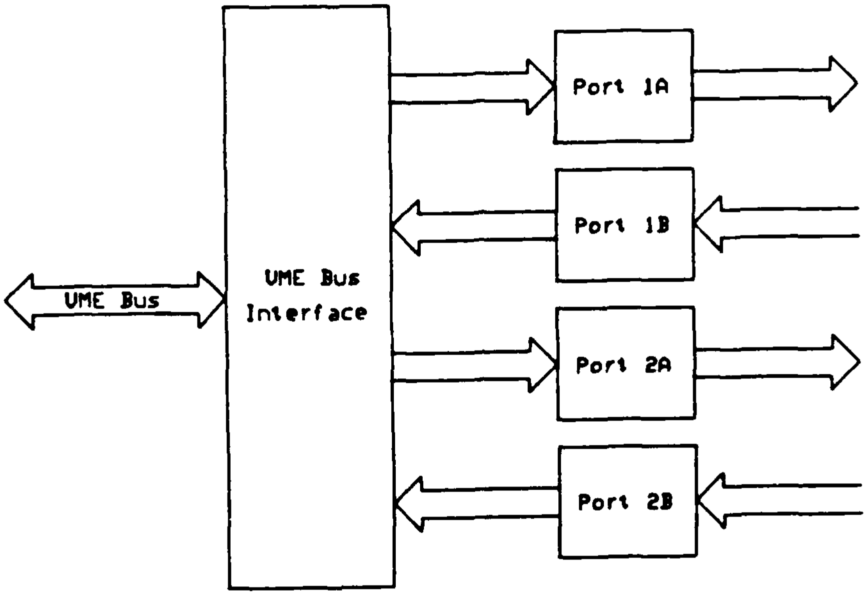


Figure 4-18. Simplified block diagram of the Parallel Port Board.

Design Tradeoffs

Many of the control processors have to exchange information. For example, the SCU informs the MCs about jobs to be run, the MCs request file service from the MMS, and the MMS directs the MSUs to perform file accesses. The Parallel Port Board provides a simple bidirectional connection for this exchange of control information. As shown in Figure 4-8 the links based on the Parallel Port interconnect various processors in the system. Two basic tradeoffs were analyzed to arrive at the current design: (a) whether the board should be connected to the VMEbus or to the private bus of the CPU, the I/O channel, and (b), which type of communication link should be used.

While a connection to the VMEbus results in a board that can be used with any type of VMEbus CPU Board, it also requires a VMEbus slot for every Parallel Port Board. In the MMS, three CPUs share the same VMEbus, and they could thus not access their Parallel Port Boards without competing for the VMEbus mastership. The I/O Channel, on the other hand, is independent from the VMEbus. A Parallel Port Board connected to the I/O Channel thus needs no VMEbus slot, and it is well suited for the MMS application. Therefore the Parallel Port Board was connected to the I/O Channel.

The communication link should provide a simple bidirectional link. It will be used to exchange control information. Serial and parallel links were considered. A serial connection has the advantage of requiring few lines, but the data rate of serial port chips is limited. Chips like the MC6854 or the MC68681 support a maximum data rate of 1M bit/second, equivalent to 125K byte/second. One significant drawback of serial connections is the missing handshake. A transmitter can send data along the serial link but has no knowledge whether the data was received by the destination or lost during transmission.

A parallel link uses more signal lines than a serial connection, but it can easily achieve the higher data rate. The data rate of the MC68230 parallel port chip, for example, is in excess of 500K byte/second. The parallel link implemented by an MC68230 parallel port chip also provides interlocked handshake capability. This guarantees that data is transmitted by the source only if the receiver is ready to accept them. In the final design, the parallel link with an MC68230 chip was chosen due to the higher data rate and the handshake capability.

Functional Description of the Parallel Port Board

A block diagram of the Parallel Port Board is shown in Figure 4-19. The private I/O bus of the MVME 110 CPU Board, the *I/O channel* is connected to the Parallel Port Board's *I/O Channel Interface.* This interface is connected to the two parallel ports *Port 1* and *Port 2,* and the parallel ports are connected to the outside world through bidirectional drivers *DRV1 - DRV4.*

The I/O Channel Interface buffers the data, address, and control lines of the I/O Channel. It detects an access to the board if the access strobe IOC.STB* is asserted, and bits IOC.A8-A11 of the address bus are identical to the board's base address. The board base addresses is set by four switches. The chip select CS1* for PORT 1 is asserted if A6 is '0,' and the chip select CS2* for PORT 2 is asserted if A6 is '1.' The read/write signal IOC.R/W* is forwarded to Port 1 and 2, determining the direction of the data transfer. Data is passed to and from the ports via the data lines D0-D7. Address lines

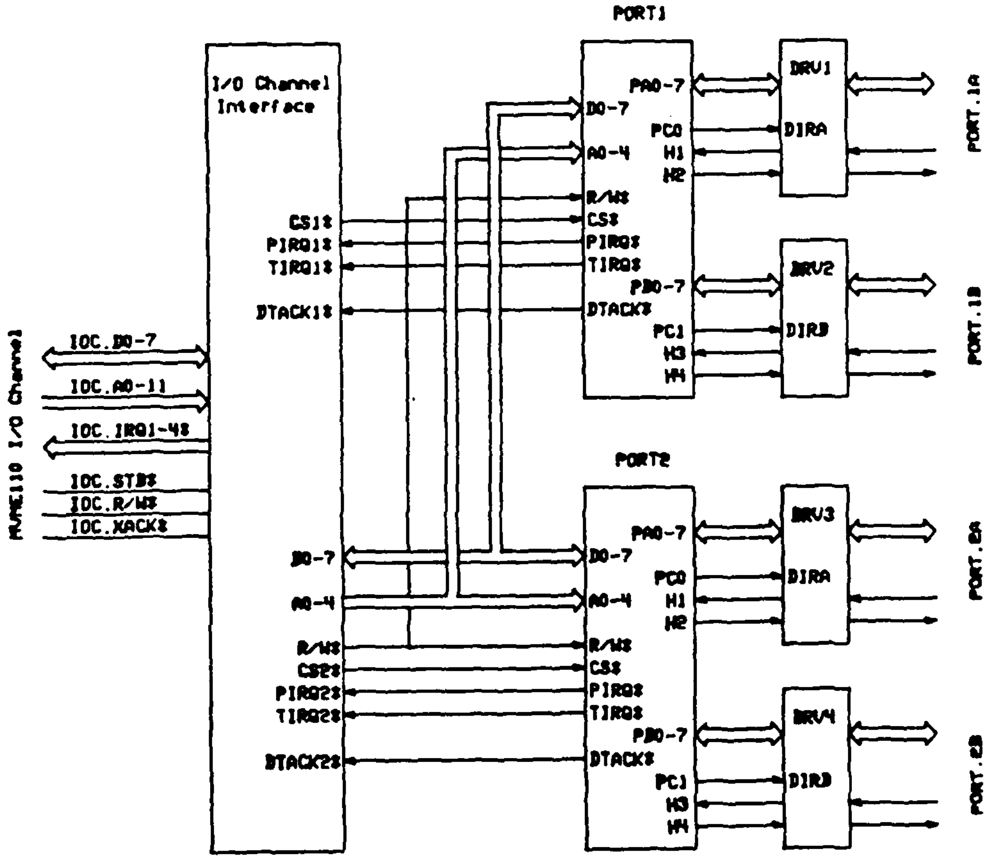


Figure 4-19. Detailed block diagram of the Parallel Port Board.

A0-4 determine which of the internal registers of the ports is accessed. As soon as a port has completed the access, it passes back an data transfer acknowledge signal (DTACK1* for Port 1 and DTACK2* for Port 2). The port DTACK* signals are combined and sent to the CPU as IOC.XACK*. When the CPU receives the signal, it negates the access strobe which causes the ports to negate DTACK*, thus completing the access. The two parallel ports can generate four interrupts which are forwarded to the I/O channel interrupt lines IOC.IRQ0-3*.

The input/output port lines PA0-7 and PB0-7 of Port 1 and Port 2 are routed through bidirectional drivers to the outside world (i.e., a flat cable connector). Output lines PC0 and PC1 are used to determine the direction of the data transfer, where PC0,1 = '0' sets the drivers to output, while PC0,1 = '1' sets them to input. Four handshake lines H1-4* are provided to control data transfer operations.

4.3.7 The Network Interchange Box Boards

General Description of the Network Interchange Box Boards

The PASM Extra-Stage Cube network is implemented by three types of boards: a Input Stage Board, and Intermediate Stage Board, and an Output Stage Board. Input and Output Stage Boards are connected to Network Interface Boards in the PEs through flat cables.

Simplified block diagrams of the three board types are shown in Figure 4-20a, b and c. Each contains a *2-by-2 Interchange Box and Routing Controller*. The Input and Output Stage Boards also contain multiplexers *(MUX)* that

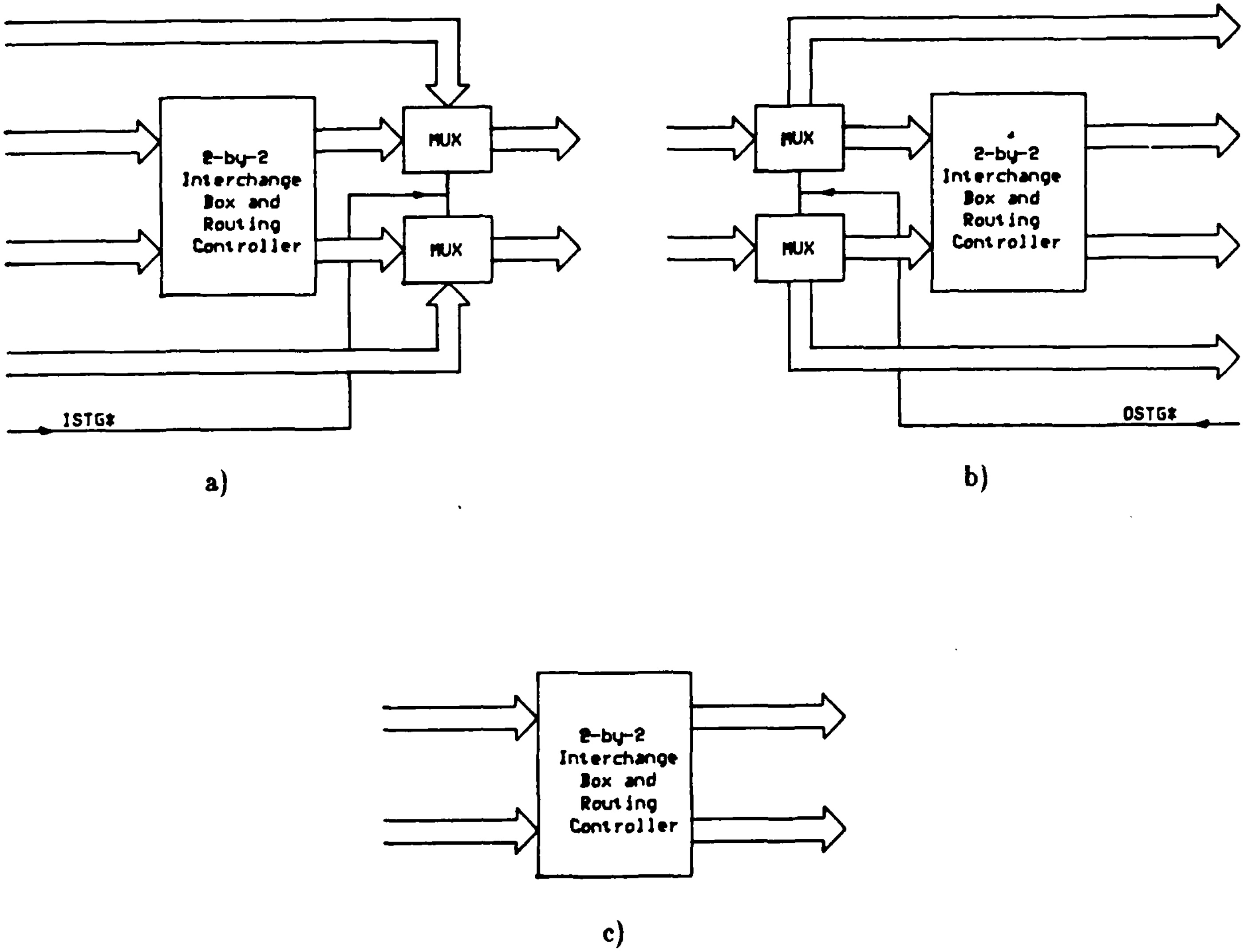


Figure 4-20. (a) Simplified block diagram of the Network Input Stage Box Board. (b) Simplified block diagram of the Network Output Stage Box Board. (c) Simplified block diagram of the Network Intermediate Stage Box Board.

facilitate bypassing of the Interchange Box.

Design Tradeoffs

The fundamental design of the prototype interconnection network was not done by this author, and design tradeoffs relating to the fundamental design can be found in [Dav85]. Due to design flaws in the implementation the original version, a redesign of the network was required, and this author was involved in that process.

The original design attempted to use only one board, populated differently, for input, output, and intermediate stage. Due to design flaws, this board design had to be discarded, and analysis of the wiring showed that three board types were advantageous. The layout of these three boards proved to be easier than that of one general purpose board, and the physical board dimensions could be reduced. The boards are now of the same size as VMEbus boards and thus VMEbus card cages could be used to house them.

Functional Description of the Intermediate Stage Board

A detailed block diagram of the Intermediate Stage Board is shown in Figure 4-21. The board has two inputs: the *Upper Input UI,* and the *Lower Input LI.* These inputs are connected to the *Upper Output UO* and the *Lower Output LO* through four sets of bus drivers: *Driver UI-UO, Driver LI-UO, Driver UI-LO,* and *Driver LI-LO.* The busses consist of the incoming and return signals. The incoming signals are the data lines D0-15, the parity bits P0 and P1, and the control signals PRQ* and H2*. Return signals are the control signals PGR* and H1*. The *Path Setup Logic (PSL)* determines which drivers are

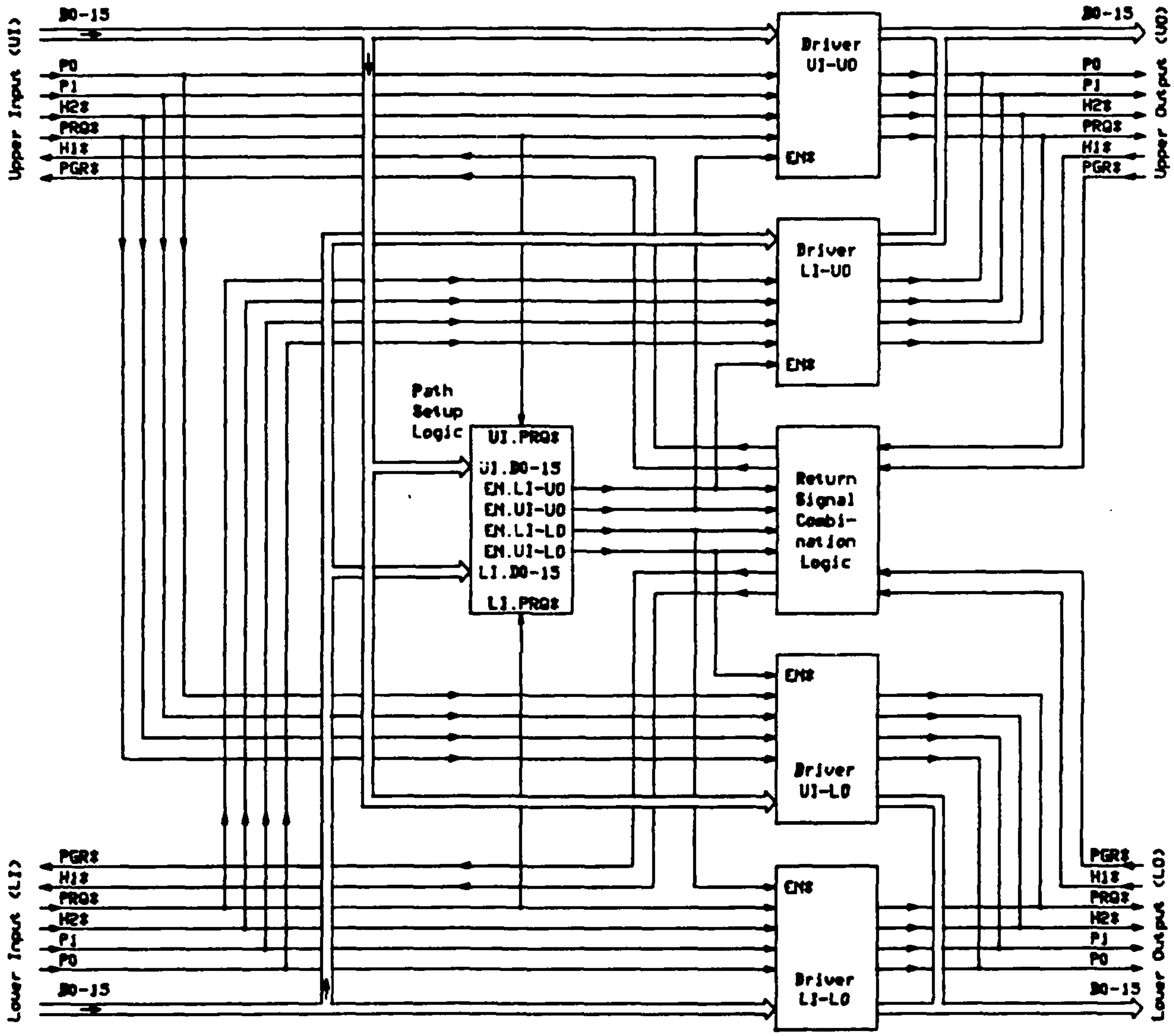


Figure 4-21. Detailed block diagram of the Intermediate Stage Board.

enabled, and the *Return Signal Combination Logic (RSC)* combines and drives signals that are passed back from outputs to inputs.

The PSL tries to establish a path as soon as it detects the assertion of either or both of the path request signals UI.PRQ* and LI.PRQ*. An Intermediate Stage Board at stage i uses bit i and bit i+8 of the data bus as broadcast and routing bits, respectively. Thus, a box at stage 3 uses bit 3 as the broadcast bit, and bit 11 as the routing bit. If no conflict with previously established pa^t^hs occurs, the PSL enables the appropriate drivers and thus propagates the message to the next stage. If a conflict is detected, the PSL will continuously try to establish the path until it either succeeds (the blocking path is dropped) or until PRQ* is negated. The PSL maintains a path, once established, until the path request is negated. Then it negates the driver enable lines for that path, and the next stage will also receive a negated path request. The PSL forwards all enable bits to the RSC. The RSC receives H1* and PGR* from the output ports and combines them. The combination is especially important if an input is broadcasting.

The Intermediate Stage Board does not alter any signal. It uses data bits and the path request lines but passes these and all other lines on to the next or previous stage unchanged. Thus the net e^ff^ect of the board is a signal delay.

Functional Description of the Input Stage Board

The Input Stage Board contains the same elements as the Intermediate Stage Board, but has in addition bypass circuitry. A detailed block diagram is shown in Figure 4-22. If the input stage enable signal ISTG* is asserted, the box functions the same way as an Intermediate Stage Board. If ISTG* is

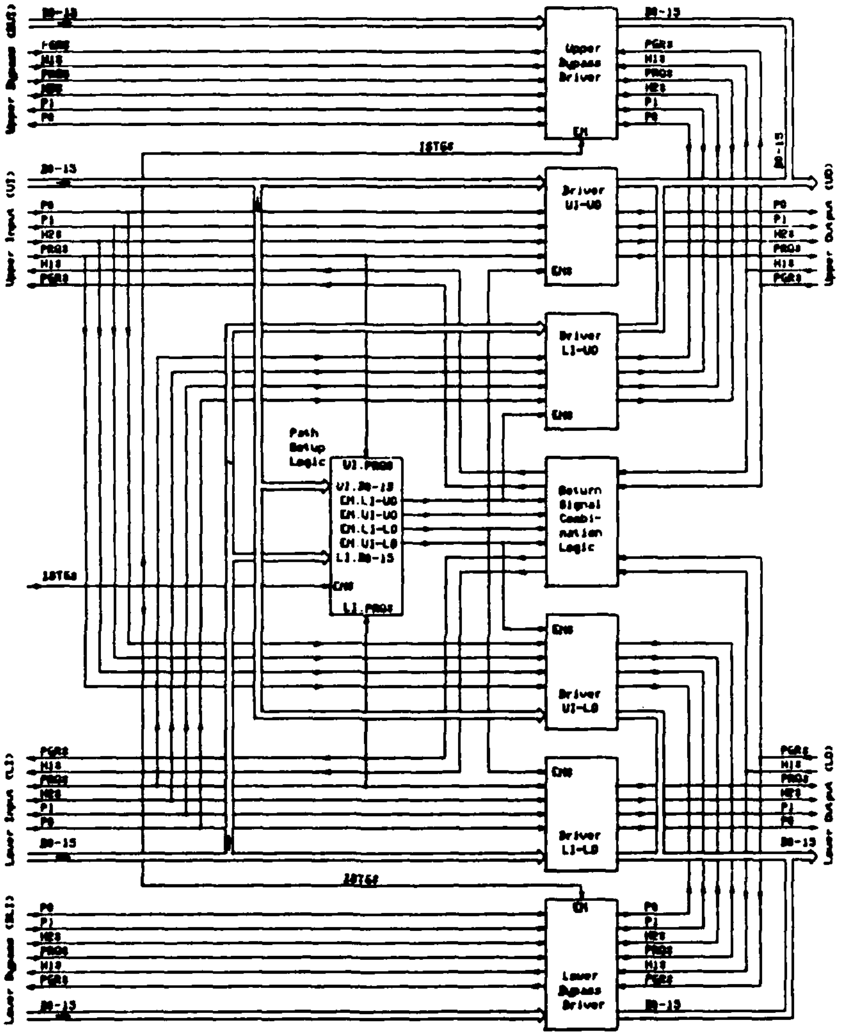


Figure 4-22. Detailed block diagram of the Input Stage Board.

negated, the lower bypass input BLI is routed to the lower output LO, and the upper bypass input is routed to the upper output UO. Thus the interchange box itself is bypassed, and faults in it can be circumvented.

Functional Description of the Output Stage Board

The Output Stage Board contains the same elements as the Intermediate Stage Board, but has in addition bypass circuitry. A detailed block diagram is shown in Figure 4-23. If the output stage enable signal OSTG* is asserted, the box functions the same way as an Intermediate Stage Board. If OSTG* is negated, the lower input LI is routed to the lower bypass output BLO, and the upper input UI is routed to the upper bypass output BUO. Thus the interchange box itself is bypassed, and faults can be circumvented.

4.3.8 The MC/PE I/O Board

General Description of the MC/PE I/O Board

MC/PE I/O Boards perform a variety of functions in the MCs and the PEs. Since many of these functions are identical or very similar on the board residing in the MCs and the one in the PEs, only one printed circuit board had to be developed. The boards are populated differently for use in MCs and PEs. Board^s populated for use in MCs will be called MC I/O Boards, and boards populated for use in PEs will be called PE I/O Boards.

Figure 4-24 shows a simplified block diagram of the MC I/O board, and Figure 4-25 shows a simplified block diagram of the PE I/O board. The

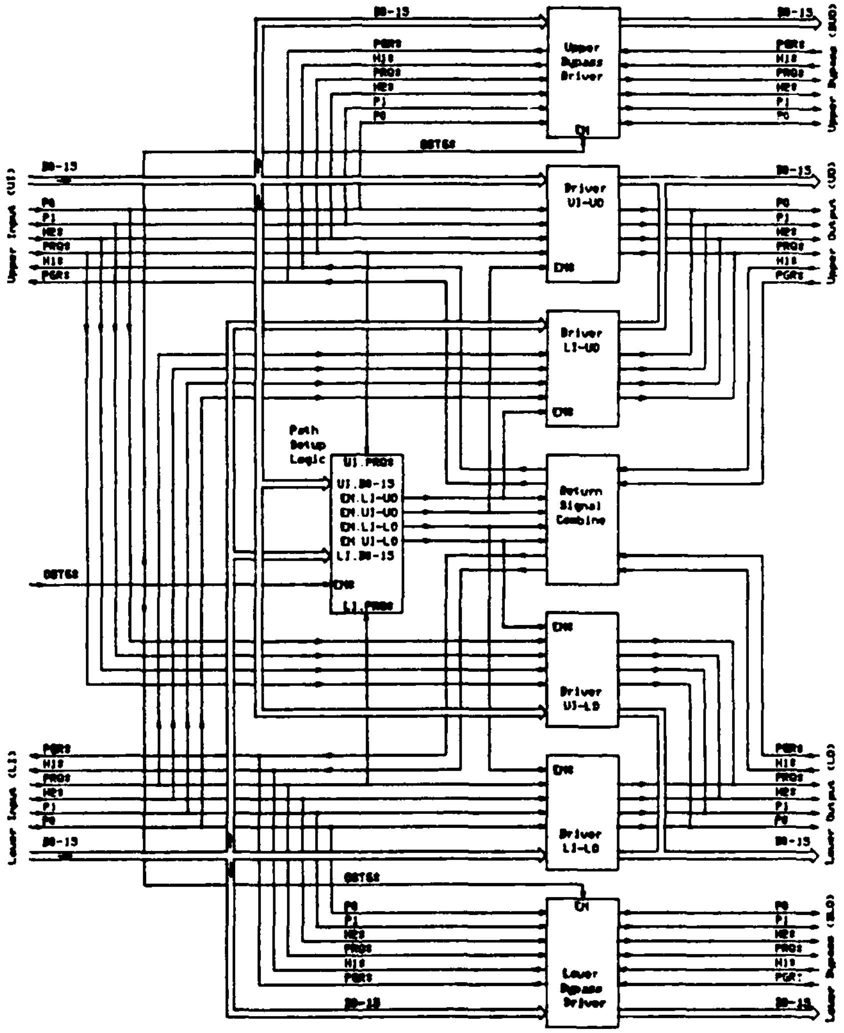


Figure 4-23. Detailed block diagram of the Output Stage Board.

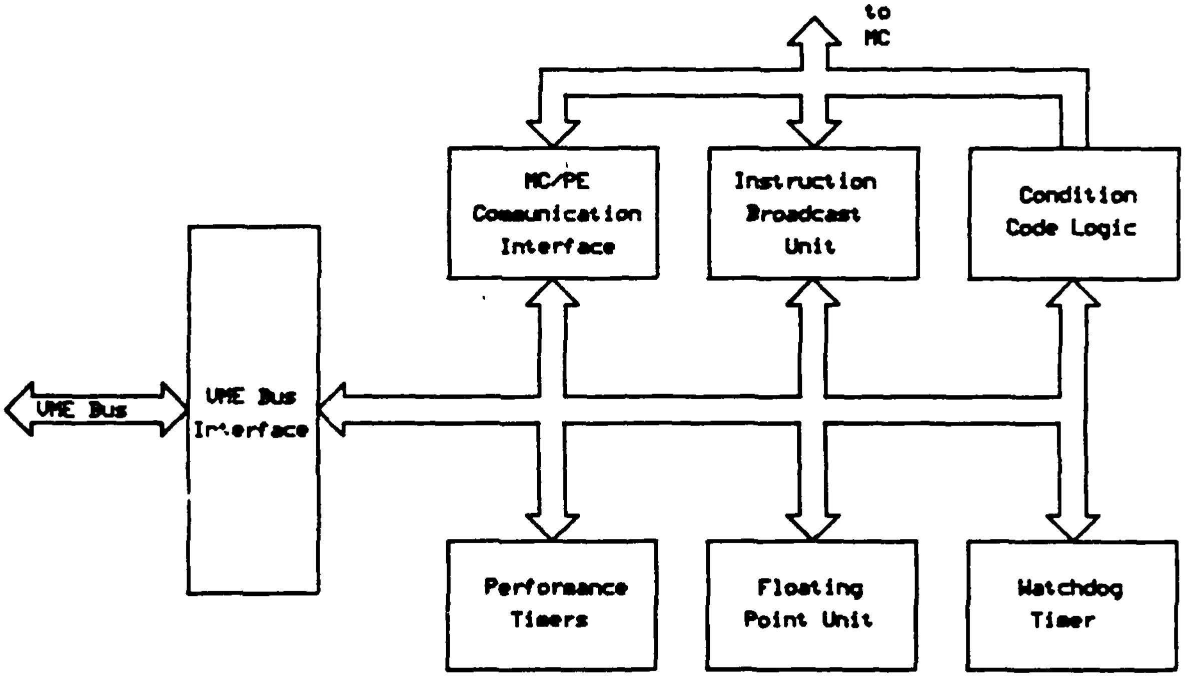


Figure 4-24. Simplified block diagram of the PE I/O Board.

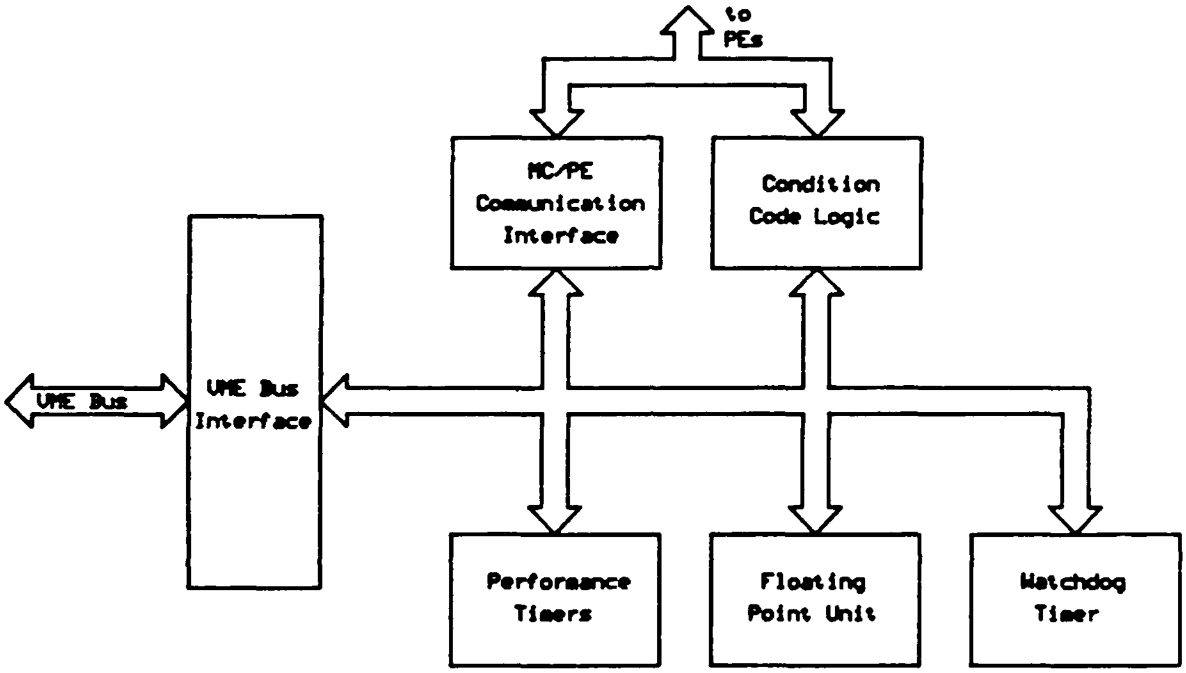


Figure 4-25. Simplified block diagram of the MC I/O Board.

similarities of the two boards are obvious. All functional units in the boards will now be overviewed, and the differences between the PE and MC boards will be pointed out.

The CPU accesses the board via the VMEbus through the *VMEbus Interface*. The interface decodes board accesses and handles interrupt requests and acknowledges. Two *Performance Timers* allow the programmer to benchmark various system activities. A *Floating Point Unit* provides hardware floating point support. The *Condition Code Logic* facilitates checking of the PE condition code by the MCs. The Condition Code Logic is sending code bits on the PE I/O Board, and receiving code bits on the MC I/O Board. The *MC-PE Communication Interface* is implemented as a GPIB standard bus. The PE I/O Board has Talker/Listener capability only, whereas the MC I/O Board has controller capability in addition to the Talker/Listener capability. The *Instruction Broadcast Unit* exists only on the PE I/O Board. It receives SIMD instructions from the MCs Fetch Unit and forwards them to the CPU. It also handles enabling and disabling of the PE in SIMD mode. The *Watchdog Timer* provides a software-controlled bus error timeout in case of erroneous accesses. In the PE I/O board, the timeout duration is different for SIMD and MIMD mode of operation.

Design Tradeoffs

One component of the MC/PE I/O Board that required detailed tradeoff analysis is the MC-PE Communication Interface. This communication channel is used to exchange control information between an MC and its associated PEs. Typical information is job scheduling information and file service information.

Several links types were considered: (a) serial links, (b) dedicatєd parallel links, and (c) shared bus.

While serial ports require few lines, their speed is limited, and they usually provide no control handshake. Two alternatives with serial connections exist: dedicated serial lines for each PE-MC connection could be used, or a serial ring can be constructed. Both alternatives were not used due to their low speed and missing handshake capability.

Dedicated parallel links between each PE and its MC are fast, have a control handshake, but they require a large number of connections, each connection consisting of a parallel link. The number of connections becomes large when the number of PEs is. large, and the cable bulk and cable connections become an implementation problem. Thus this method was discarded.

The shared bus approach has several advantages: it is scalable to a large number of PEs per MC since it uses only one bus instead of multiple connections between MC and PE; the MC can broadcast information to multiple PEs; and, depending on the implementation, the speed can be very high. A disadvantage is the difficulty to implement a physically long high-speed bus. This problem can be solved by utilizing the commercially available GPIB bus. Chip sets are available for this bus that guarantee a functioning shared bus several feet in length, and thus a GPIB bus connection was used as MC-PE communication link.

In the original design of the MC/PE I/O Board, a DMA controller was included. The DMA controller could speed up transfers through the interconnection network, and it is useful for memory-to-memory transfers. During the implementation phase it was determined that the DMA controller with the appropriate support chips could not be fitted onto the MC/PE I/O Board.

Since the Network Interface Unit as discussed in Chapter 5 that is being designed for the prototype can also perform DMA operation, the DMA controller was excluded from the design.

Functional Description of the MC/PE I/O Board

A detailed block diagram of the MC I/O Board is shown in Figure 4-26, while Figure 4-27 shows the PE I/O Board. The diagrams contain the same functional units as the simplified block diagrams in Figures 4-24 and 4-25, but they also show the control lines between functional units. In the following discussion, each unit will be discussed. Differences between units on PE I/O boards and MC I/O Boards will be pointed out.

The *VMEbus Interface* buffers all address, data, and control lines needed internally on the board and generates four and eight MHz clock signals that are required on the board. It detects an access to the board if the address modifiers VME.AM0-5 indicate an I/O space access, the address strobe VME.AS* and either or both of the data strobes VME.DS0 and VME.DS1 are asserted, and the address lines VME.A11-15 coincide with the board address which is encoded in a PAL. Depending on VME.A8-10, one of the chip select lines CS0-6* is asserted, and the appropriate functional unit is enabled. The read/write line VME.R/W* determines the direction of the data transfer. If high, data is read by the CPU, otherwise it is read by a functional unit. After the selected unit has completed the access, it asserts its data transfer acknowledge line DTACK.<unitname>*. The VMEbus Interface forwards this signal to the VME.DTACK* line. When the CPU receives the asserted DTACK*, it negates address and data strobes, causing the VMEbus Interface

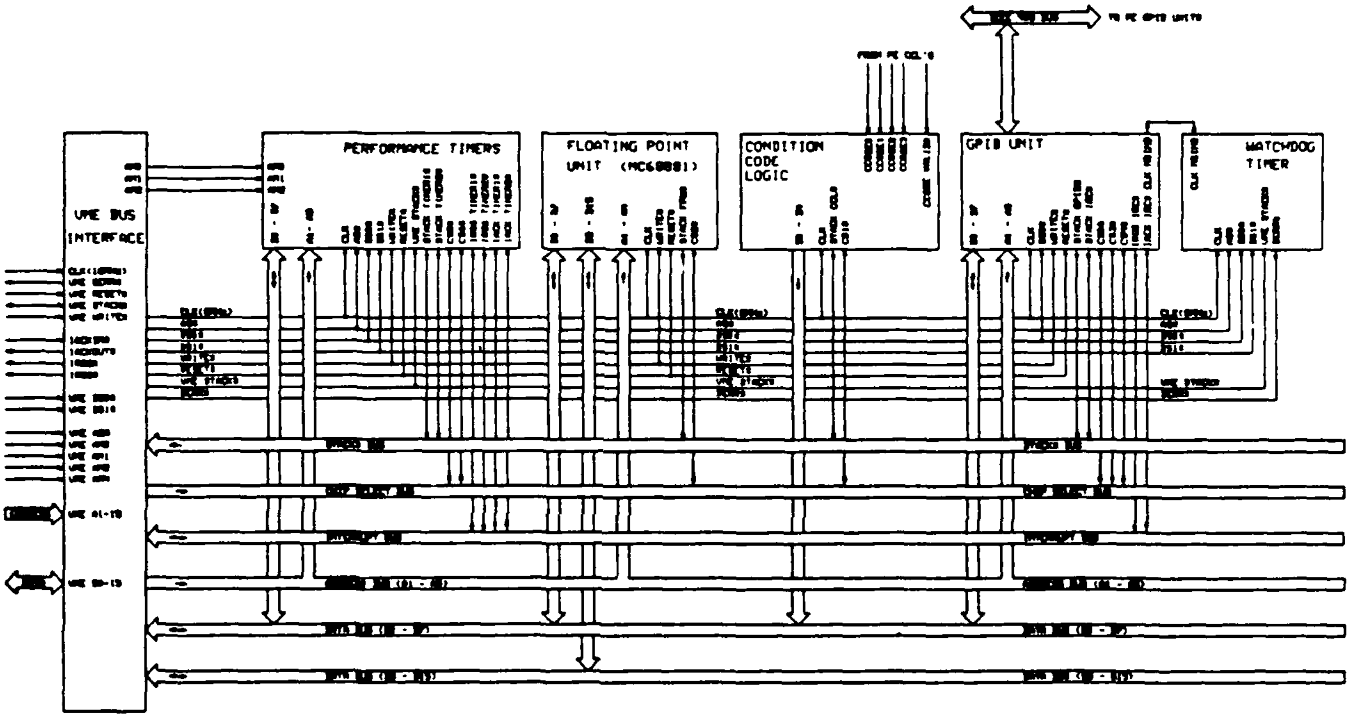


Figure 4-26. Detailed block diagram of the MC I/O Board.

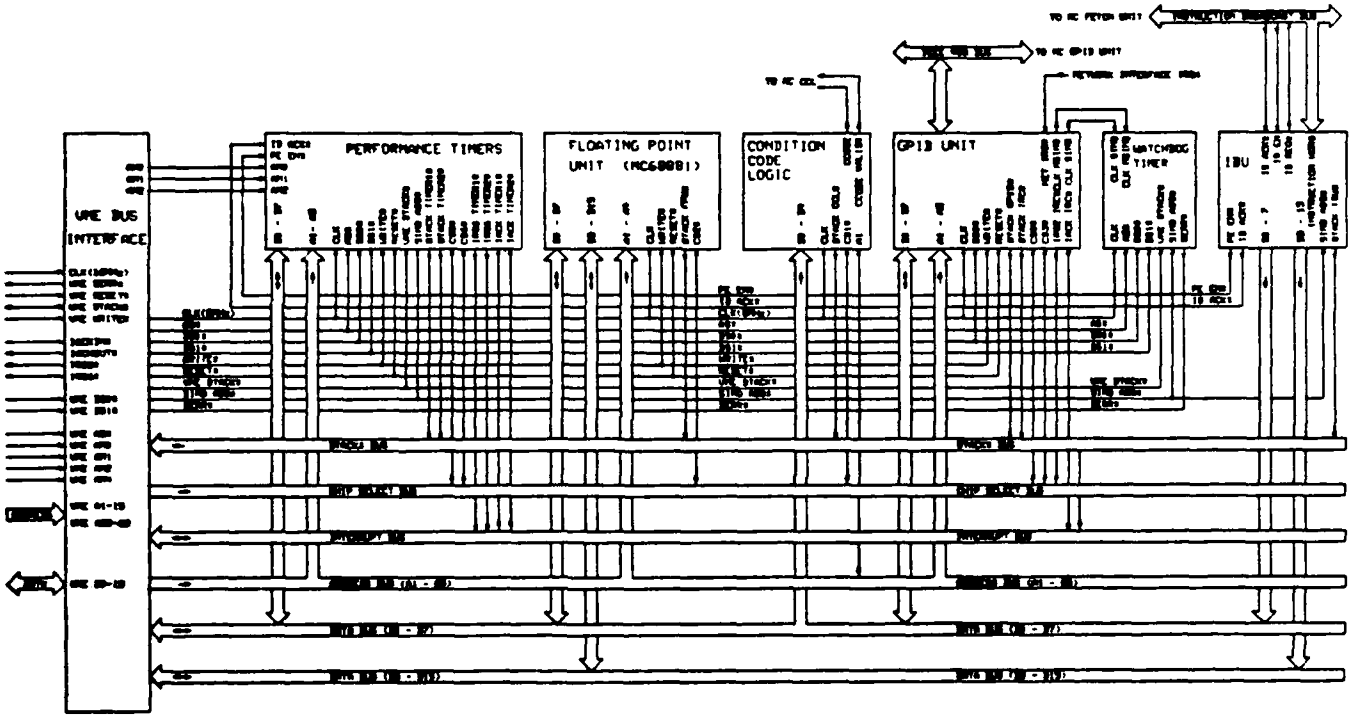


Figure 4-27. Detailed block diagram of the PE I/O Board.

to negate the chip select line. The unit in turn negates its DTACK*, and the access is completed.

The PE I/O Board also responds to SIMD instruction requests. Such a request is detected by the VMEbus Interface if AM0-5* indicate a standard memory access, VME.AS*, VME.DS0*, and VME.DS1* are all asserted, indicating a 16-bit access, VME.R/W* signals a read operation, and VME.A22 is low and VME.A23 is high. The interface then asserts SIMD.ADD*, and this signal is forwarded to the Instruction Broadcast Unit (IBU). When the IBU has received an SIMD instruction from the MC, it asserts its data transfer acknowledge line DTACK.IBU*. The CPU negates data and address strobes, causing the interface to negate SIMD.ADD*. The IBU then completes the access by negating DTACK.IBU*.

The VMEbus Interface also handles interrupts. When a device on the board asserts its interrupt request line, the interface forwards this request to the CPU by asserting either VME.IRQ2* or VME.IRQ6* which cause an interrupt on level 2 or 6, respectively. The CPU responds to interrupts by performing an interrupt acknowledge cycle. During this cycle, it asserts VME.IACKIN* and indicates the interrupt level it responds to on the address lines VME.A1-3. If no interrupt on this level was pending on the board, the VMEbus Interface asserts VME.IACKOUT* as long as VME.IACKIN* is asserted. If an interrupt is pending, it forwards the acknowledge to the proper device, and the device responds by placing an interrupt vector on the data bus and asserting the data strobe. The CPU reads the vector number and negates VME.IACKIN* which terminates the interrupt cycle.

Performance Timers. Two MC68230-based timers are included to evaluate the performance of the prototype. A number of control signals are routed to a PAL which can combine them to facilitate counting of various events. Such events can be memory cycles, SIMD instruction requests, read or write cycles, etc. Two 32-bit counters are provided, and they can interrupt the CPU on level 6.

Floating-Point Unit. An MC68881 Floating-Point Coprocessor is included on the board to speed up floating point operations. Due to budget constraints no such chips are included in the current prototype version.

Condition Code Logic (CCL). The CCL of the PE I/O Board is used to pass results of data conditional operations to the PE's MC. It contains two 4-bit registers: the condition code register (CCR) and the select register (SR). In the idle state, an invalid condition code is written to the CCR which causes the negation of CCODE.VALID*, indicating to the MC that the condition code is invalid. CCODE.VALID* is a wired-or line, and thus it is asserted only if all PEs of an MC group have asserted it. To pass a condition to the MC, the PE CPU first writes a code selecting a specific logical combination of condition code bits (e.g., less than or equal to zero) to the SR. Then it writes its condition code to the CCR, thus at the same time passing the correct condition code bit to the MC and asserting CCODE.VALID*. The MC monitors the CCODE.VALID* line, and reads the condition code as soon as the signal is asserted. After the MC has read the code, it instructs the PEs to negate CCODE.VALID* again by writing an invalid condition code to the CCR.

On the MC I/O Board, the Condition Code Logic consists of a read-only port from which the CPU can read the individual condition code bits of its PEs and the CCODE.VALID* signal.

GPIB Unit. A GPIB link provides communication between an MC and its PEs. The PEs have Talker/Listener capability, and the MC has additional

GPIB Controller capability. An MC68901 Interrupt Controller handles the interrupts generated by the GPIB chips and an interrupt line from the Network Interface. Data Manuals for the Intel 8291 (Talker/Listener), 8292 (Controller), and 8293 (Transceiver) detail the operation of the link.

Watchdog Timer. The VMEbus operation is asynchronous, and thus a watchdog timer is required that terminates erroneous bus accesses. The Watchdog Timer has two modes of operation. If the access is not to the SIMD Instruction Broadcast Unit, and is not terminated 125μs after it has been initiated, the Watchdog Timer asserts the bus error signal BERR*. The CPU will then terminate the access and execute a bus error routine. On a PE I/O Board, if the Instruction Broadcast Unit is accessed, the Watchdog Timer asserts the bus error after one second has expired. This long duration is necessary since a PE could be disabled for a long time during SIMD programs, and thus such a long bus cycle is legitimate. To achieve flexibility, both timeout intervals are programmable and are not restricted to the values given above.

Instruction Broadcast Unit (IBU). The IBU exists only on the PE I/O Board. When the CPU requests an SIMD instruction, the VMEbus Interface asserts SIMD.ADD*. The IBU then requests an instruction from its MC by asserting the request signal IB.REQ* that is forwarded to the Fetch Unit. As soon as the IBU receives an asserted enable bit IB.EN and an asserted instruction broadcast acknowledge IB.ACK*, it asserts it data transfer acknowledge line DTACK.IBU*. When the CPU has received the SIMD instruction, it negates its address and data strobes, and the VMEbus Interface negates SIMD.ADD*. When detecting a negated SIMD.ADD*, the IBU negates IB.RQ*. The Fetch Unit responds by negating IB.ACK*, the IBU negates the data transfer acknowledge, and the access is completed.

4.4 Mechanical Construction of the PASM Prototype

On of the issues least considered during the early phases of the design of many systems is the mechanical construction. When actual hardware design and construction is begun, however, many issues surface that are not related to digital or logic design but are of a mechanical nature and are just as important as working logic. Some examples that were experienced during the PASM prototype construction are given below:

* Originally, the PASM PE CPU was anticipated to be designed and manufactured in-house. Consequently, no standard card format was chosen, and it was very difficult to find a suitable industrial card cage. After it was decided to use the VMEbus for all system processors, card cages were easy to obtain.

* Due to changes in some board types which reduced their space requirements, the system size could be reduced from five cabinets to three cabinets. This greatly reduced wiring complexity and cable length, and also facilitated a ground system with very low resistance.

* The prototype contains 16 card cages, each of which have to be supplied with power and ground. Until the current solution utilizing copper bus bars was found, various ways to connect power and ground were devised which had to be discarded due to mechanical stability reasons or excessive cable lengths.

* Since the prototype will consume about four kW of electrical power, forced air cooling is required. The individual card cages had to be arranged carefully so that air can flow through all cages. For this reason, for example, all disk drives are placed in the bottom of the cabinets and not close to the disk controllers which are close to the top. Placing the disks at the

top of the cabinets would have impeded air flow.

These considerations and many others led to the current mechanical construction of the prototype. Figure 4-28 shows a diagram of the prototype racks. The system is built from three 19" racks with no separating walls. The top six card cages contain the IOP, SCU, CS, MMS, and the MSUs. Below these components are the MCs (in the middle rack), and their associated PEs (to the left and right of the MCs). Below the PEs are the interconnection network and the disk drives. The central ground bus bar, which guarantees very low-resistive ground connections between all points of the system, and the AC power distribution are in the center rack. The power supplies of the system are mounted in the back and on the bottom of the cabinets.

4.4.1 Cooling Requirements

Preliminary calculations show an estimated worst case power dissipation of 4 kW in the prototype. In this section, the air flow required to cool the system is determined. The ambient room temperature of the PASM laboratory in which the system is housed is assumed to be 20 ° C. The cooling air is assumed to have an exit temperature of 40 ° C which is significantly below the maximum free air operating temperature of 70 ° C which is specified for TTL integrated circuits. Thus a safety margin for higher room temperature and local hot spots is given. The volume of air per second needed to cool 4 kW is given by

$$V = \frac{Q}{\Delta T \, c_p \, \rho_{air}}$$

where Q is the heat dissipated in one second (4000 J), ΔT is the temperature difference (20 ° C), c_p is the specific heat of air at constant pressure (the

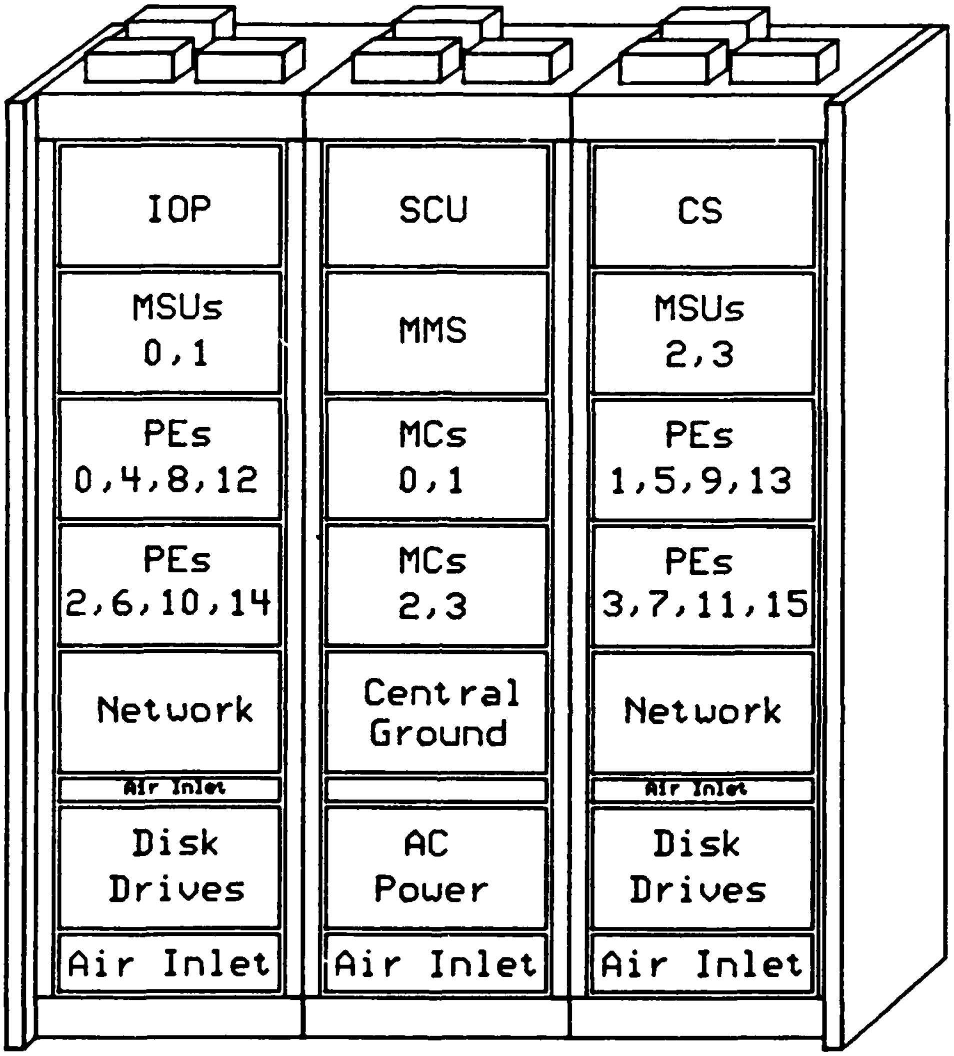


Figure 4-28. Mechanical construction of the PASM prototype.

pressure change due to the cooling fans is negligible), and ρ_{air} is the specific density of air. From tnis equation, an air movement of 0.16m^3 per second is calculated. Fan manufacturers commonly specify the air flow in cubic feet per minute (CFM). If this unit is used, 330 CFM are required. In the prototype, nine fans are used that are capable of moving a total of 1000 CFM at a static pressure of 15 mm of water. The fans used are thus sufficient to cool the machine and have enough safety margin to allow for higher static pressure or uneven airflow. The fans have been placed on top of the cabinets in such a way that air moves through all card cages as evenly as possible.

4.4.2 AC and DC Power Distribution

In this section, the reasoning behind the AC and DC wiring is shown. The prototype dissipates approximately 4 kW of power. Since it is connected to the 110V net of the university, a maximum current draw of 40A can be expected. Since switching power supplies are used, the in-rush current at power-on time will be much higher. If is therefore not advisable to switch on all of the system at the same time since this might result in power failures. Therefore a power-up sequencer was developed that contains six power relays that are activated in one-second intervals. For example, the cooling fans are activated first, followed by the power supplies for the control processors (SCU, IOP, CS, MMS), etc. The power-up sequencer also assures that the system stays switched off after a power failure. This allows the operator to make sure the power failure did not cause any damage to the system before the system is restarted.

The DC power system needs careful design as well. For example, a single power supply provides all 5V-current needed by two MCs and their eight associated PEs. In worst case, this current can amount to 150A. If a cable or contact would have a resistance of only 10 milliohms, a voltage drop of 1.5V would result. Such a voltage drop is of course intolerable in digital logic circuits. Therefore large gauge wire was used for all 5V lines, and the ground system is even less resistive through the use of massive (1" by 1") copper bars. Figure 4-29 show details of the DC wiring in the prototype.

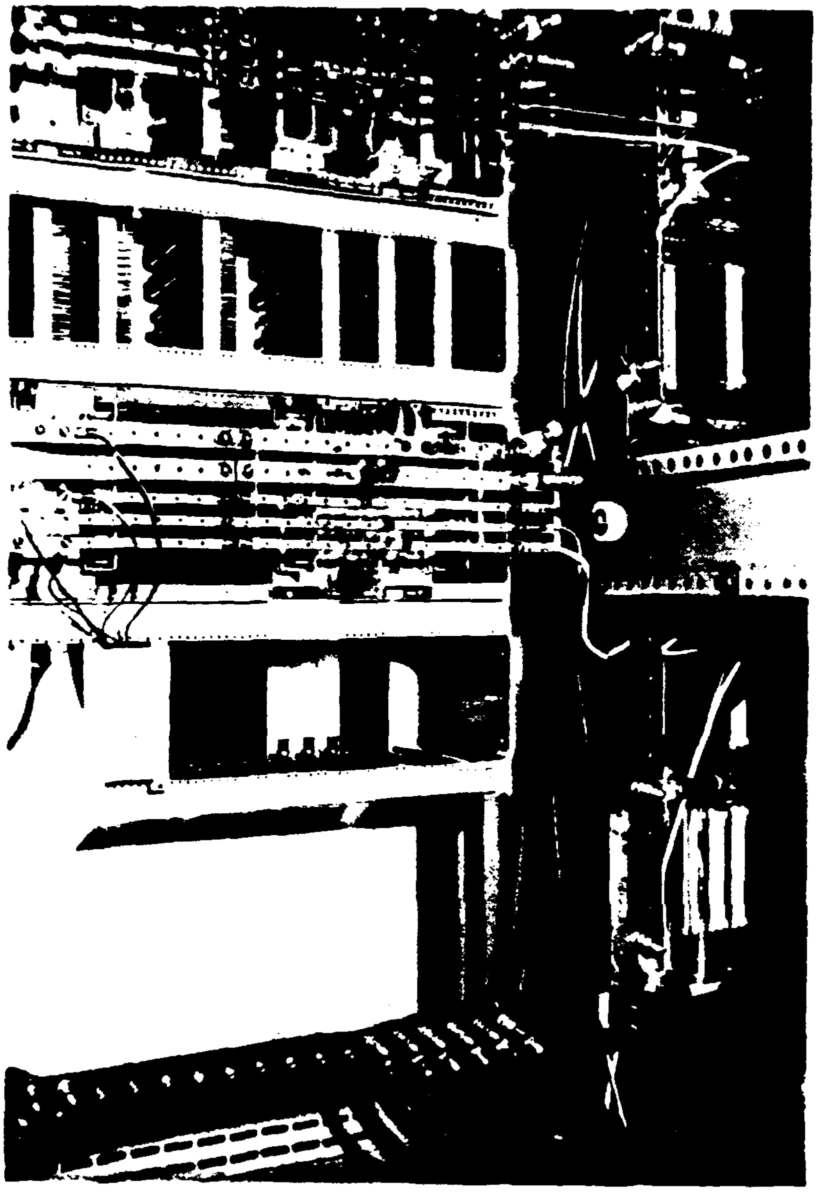


Figure 4-29. Detail of the DC wiring in the prototype.

CHAPTER 5

THE NETWORK INTERFACE UNIT

The PASM parallel processing system has a PE-to-PE structure, i.e., each PE is a processor-memory pair, and the PEs exchange messages through the interconnection network. This is in contrast to a parallel processor in processor-to-memory organization where the network is used to pass shared-memory accesses. While a shared memory access implicitly contains the interconnection network routing control information in form of its address, the message passing scheme requires the routing to be specified explicitly. Also, any message has to be moved from a source processor's memory or registers to the network, and the destination processor has to move the arriving message from the network to its memory or its registers. The overhead incurred by these tasks decreases machine performance.

In this chapter a *Network Interface Unit (NIU),* which is one way to decrease the overhead, is presented. First, assumptions about the interconnection network and the network interfaces will be stated, followed by an analysis of the overhead to send a message across the PASM network. Then a description of the network transfer method employed in the current PASM prototype will be given. The analysis of the overhead and the current approach clearly justifies the use of an NIU. One possible NIU structure and the capabilities of the design will then be presented.

5.1 Communication Overhead Analysis

5.1.1 Assumptions about Network and Network Interface

Since the purpose of this analysis is the study of different types of network interfaces (passive vs. intelligent), it must be independent of the structure of the interface. Thus no assumptions about the interface are made other than that it provides signals compatible with the actual network implementation. Therefore the analysis is valid for a passive interface as implemented in the current prototype as well as for the Network Interface Unit as proposed in this chapter.

The network structure is given by the PASM architecture, and thus assumptions about the network can and have to be made. For example, if the network were very slow, a very slow interface would suffice; if it is fast, a fast interface is needed to utilize it fully. For this discussion, the Extra Stage Cube Network [AdS82] as implemented in the prototype is assumed, and the prototype network parameters [DaS85b] are used. This network is circuit-switched, uni-directional, 16 bits wide, and transfers two parity bits with each data word. Data transfers are controlled by an interlocked handshake of two control lines. The propagation time of a data word is approximately 80ns per stage (including control signal propagation). For the prototype network, this results in a total delay of 400ns for a transfer. This time is roughly identical to the memory access time of a Motorola MC68010 CPU which is used in the prototype.

A path through the network is established by specifying a routing and a broadcast tag, and asserting a PATH REQUEST signal. The tags travel from stage to stage until either a blocking interchange box is encountered or a destination processor is reached. The path request remains pending at a blocking interchange box until either PATH REQUEST is negated or until the box becomes available, at which time the message passes through the box on to the next stage. The path stays established until PATH REQUEST is negated. This is true for complete paths as well as partially established ones. When a PATH REQUEST has reached a destination PE, a PATH GRANT signal is sent back to the source PE. It takes approximately 200ns per stage to establish a path if no blocking occurs. Thus, a path in the prototype network is established in approximately 1000ns.

In systems like MPP [Bat80, Bat82], user programs access the interconnection network directly by writing to or reading from network input/output registers. In PASM, users in general will employ operating system calls to transfer data through the network. This is one way to prohibit multiple users from interfering with each other's programs, and it prohibits users from changing registers controlling hardware. Message transfer operating system calls for PASM are discussed in [KuS85b]. However, operating system calls should not unnecessarily restrict the user's flexibility.

This discussion does not exclude the possibility of running PASM in single-user mode for a dedicated application. In this case, the operating system might be completely circumvented to achieve maximum performance, and all operating system related overhead is avoided. Normally, the operating system will be used, and the discussion therefore must take it into account.

5.1.2 Analysis of Overhead

When a PE needs to send a message, it calls the operating system. The operating system may or may not use a high level protocol to achieve reliable communication (see discussion below). It calculates routing and broadcast tags, sets up a path through the network, transfers the data, and relinquishes the network path. It then returns to the calling routine. A destination PE accepts messages that arrives at its network input port, and moves the message from the network to its memory. These issues will now be discussed in detail.

High-level Communication Protocol

The hardware implementation of the interconnection network provides parity and correct destination checking. It does not perform any error correction, and thus the hardware is not sufficient to ensure reliable communication. The user or the operating system is therefore responsible for providing reliable communication. The XINU operating system [Com84], which is used in a modified version in the PASM PEs, has provisions for reliable communications. However, the elaborate schemes required incur high overhead which may not always be warranted. A user might, for example, want to trade off reliable communication to higher performance or might prefer his own scheme to achieve reliability. To keep this flexibility, the high-level communication protocol will be left unspecified here, and issues regarding reliable communication will not be further considered in the discussion of network transfer overhead.

Routing Tag Calculation

Routing and broadcast tags are generally not known at compile time. The programmer of parallel programs always deals in terms of a virtual machine, and it is not determined at compile time which physical PEs will be assigned to the program at run time. The tags are generally not known at load time either. Even though the physical PEs on which a program runs are determined at load time, the programmer might have used a variable to specify a destination processor, and the mapping of the destination specified by such a variable must be performed at run time since the value - and thus the destination - is only known at run time. Thus, the calculation of routing and broadcast tags must be performed at run time, and it must be performed by the operating system since the user program is not aware of the physical PEs it is running on, and since only the operating system can enforce the machine partitioning. Calculation of the tags at run time is also necessitated by the need to route around faults if a fault occurs in the Extra Stage Cube Network. Since the operating system knows about these faults, it can accommodate the rerouting invisible to the user program.

Consider which information has to be known in SIMD programs to map the logical PE numbers that the programmer uses into physical PE numbers. First assume no fault exists in the network. In his program, the user has selected the size of logical machine is required to run his program. At load time, the operating system has determined which physical partition runs the logical machine. The partition is completely determined by the size of the partition, and the partition designator D which is the lowest numbered MC in the partition. The partion size is given by S=MN/Q, where $N=2^n$ is the number of PEs, $Q=2^q$ is the number of MCs, and M is the number of MCs combining their efforts. All PEs in the partition agree in the $q-s$ low order bit position, and the mapping from logical PE i to physical PE p is given by $p = i \times 2^{q-m} + D$. Thus only two variables, the partition size and the designator number, have to be known to calculate the no-fault routing tag from a logical destination number. Even for a system with 1,024 PEs, each of the numbers can be represented in five bits. The partition size is given by s, and $s \leq Q$ (=32). Only 32 MCs exist in a 1,024-PE system, and thus five bits are sufficient to express the designator.

Since the PASM Extra-Stage Cube is single fault-tolerant, consider the case of a single fault in the network, and assume the fault location is known. Then the partition size, designator, and fault location are sufficient to determine the routing tag through the network and route around the fault if required [Sie85].

In MIMD programs the problem of calculating routing tags is even more complex. Here the user has no notion of processors but only processes. Processes can be created at run time, and the number of concurrent processes might be determined at run time also. When a process needs to send a message to some other process, it will specify the message destination in terms of a process identifier. Since only the operating system is aware of the processor a process runs on, it alone can map a process identifier into a PE number. The operating system in a PE, however, does not keep track of all processes of the concurrent program. This information is kept in the MCs, and thus the PE has to request the mapping from process identifier to PE from its MC.

Set-up of the Network

Once the routing and broadcast tags of a message are known, a path through the network can be established. At the source, this requires moving the tags to the network, asserting the PATH REQUEST, and monitoring the PATH GRANT which is asserted by the destination PE as soon as a path has been established. If the path is not established immediately (i.e., blocking occurs), three possibilities exist [DaS85a]. The processor can drop its request and try later (drop algorithm), it can wait until the path has been established (hold algorithm), or it can wait until either the path has been established or some specified time has expired (modified hold algorithm). The drop algorithm requires either a busy wait of the PE until a new try is made, or, if multiprogramming is available, a context switch to another process. The hold algorithm requires busy wait of the PE since the PE must respond immediately when the path is established. If it would not respond at once, many other messages would be blocked unnecessarily. The modified hold algorithm requires either the setup of a timer that informs the CPU when the hold time has expired, or else a busy wait loop could be employed in which the PE tests the PATH GRANT a certain number of times.

At the destination, the PE has to assert the PATH GRANT as soon as a PATH REQUEST is received, must check whether the message was destined for it, and must read routing and broadcast tags - which are the first data items of the message - from the network. No further actions are required from the destination PE during path setup.

Data Transfer

As soon as a path through the network is established, data transfer can begin. The source PE fetches data items from its memory or its registers and sends them into the network until all necessary data have been transferred. The destination PE fetches the data items from the network and stores them in its memory or registers. Note that a PE can be sending and receiving at the same time. This is common in SIMD mode; whenever a permutation is performed, all PEs of a virtual machine send and receive. In this case, the send and receive overheads add. During the data transfer, the source must add parity bits to the data words, and the destination must check the parity. The source PE must also start and monitor a watchdog timer that ensures data that was sent into the network has reached its destination [DaS85b].

The time to transfer a data item through the network is 400ns as discussed above. In order to move a data item to the network, a PE CPU has to fetch the appropriate *move* instruction from memory and move the desired data item from memory or CPU registers to the network. Assuming a memory cycle of 400ns, this transaction takes at least two memory cycles (register to network), but can take many more (e.g., memory-to-memory transfers). If a block of data is transferred, address calculations have to be performed, and the average time to move a datum to the network becomes even longer. One way to reduce the transfer time is the use of a DMA controller. A DMA controller can read a memory location and send the value to the network every two memory cycles, and it performs automatic address increments for block transfers. A DMA controller is useful only if a large consecutive block of memory has to be transferred since any DMA operation has to be initiated by the PE CPU, and the overhead of the initialization must be justified by the time saved through faster data transfer. If not consecutive but period locations have to be transferred (e.g., every 256th byte of an array), a conventional DMA controller cannot be used and the CPU has to perform address calculations and data movement.

Thus, if a processor is only sending, data can be delivered to the network only at half the maximum speed of the network. If a processor is sending and receiving, data can be delivered at only one forth the speed. If no DMA controller is available or it cannot be used due to the nature of the transfer, the ratio becomes much worse.

Path Relinquish

After all data have been transferred, the path must be relinquished. The source PE negates its PATH REQUEST and is done, while the destination processor negates its PATH GRANT as soon as it receives the negated PATH REQUEST.

5.1.3 Analysis Results

The overhead described in the above section can be classified into three groups: (a) overhead before the network path is established; (b) overhead while the path is established; and (c) overhead after the data have been transferred. (a) and (c) have a negative effect on the overall computational speed only, but they do not affect network traffic (i.e., they do not increase blockings). (b) not only increases computational time, but also underutilizes the network, and therefore keeps paths through the network open longer than necessary. This increases effective network usage, and results in more network conflicts in

MIMD mode. In SIMD mode, no conflicts in a partition occur, but program runtime is still increased.

Thus, for the given network, the PEs and not the network are a bottleneck in inter-PE communication. Reduction of (a) and (c) are desirable to increase the computational speed of the system. Reduction of (b) will be beneficial for the computational speed, but will also utilize the network better and thus justify the expense of a fast network, and will decrease network blockings which are costly due to busy waits or context switches.

5.2 Overhead Reduction by Hardware Addition

Many different alternatives to reduce the overhead exist. If the processor, memory, and network speed are assumed to be constant, a promising alternative is the addition of hardware to offload tasks from the PEs' CPU. In this section, tasks will be identified that have to be performed by the CPU, and all other tasks will be analyzed later when the actual hardware is discussed.

In the above discussion it was shown that the high-level protocol should be optional. Thus no need for supporting hardware is immediate. If a very fast way to perform this protocol by additional hardware could be found, no penalty would be incurred by the use of such a protocol. In that case, supporting hardware should be considered. In the current network implementation, no very fast method is possible since every high-level protocol requires positive acknowledge, i.e., the destination PE must inform the source PE that it has received data. Since the network is unidirectional, a path from destination to source has to be established in addition to the path from source to destination. Apart from the overhead caused by such a path, network blockings could occur and deadlock is possible. These reasons exclude the high-level protocol from hardware support.

The calculation of routing and broadcast tags in SIMD mode requires mapping of logical PE numbers to physical PE numbers as shown above. This mapping could be performed by special-purpose hardware that is programmable by the operating system. Since this hardware can also enforce the network partitioning, the routing tag calculation in SIMD mode need not but performed but only controlled by the operating system itself. As will be shown in the discussion of the Network Interface Unit such an approach reduces communication overhead significantly. In MIMD mode, no strict mapping from process identifier to physical PE exists in general. Thus no simple one-to-one correspondence can be assumed, and the PE operating system has to perform the calculation in software.

All other tasks as analyzed above do not involve operating system variables, and thus they can be hardware supported.

5.3 Shared Memory Issues

So far only message passing from PE to PE has been considered, which is the natural mode of operation for a network with PE-to-PE organization. In the network itself, no provision for shared memory exists. Operating system research and MIMD considerations have shown that shared memory is desirable for a variety of reasons [Kue86]. For example, operating system and user programs use shared memory for synchronization, semaphores, and the sharing of results. Therefore it is desirable to utilize the current network architecture for shared memory accesses, even if such accesses may be slow.

If shared memory accesses are allowed, they have to be distinguished from other messages. An identification word distinguishing regular messages and shared memory accesses needs to be added to every message. This is overhead not considered in the above discussion.

Prototype Network Interface

The prototype supports message transfers through a passive network interface. This interface accepts data from the CPU and routes it to the network. It generates parity for messages sent into the network, and checks parity of arriving messages. It also automatically verifies whether an arriving message was destined for it. Apart from these capabilities, the interface is completely controlled by the main CPU, and no hardware support to reduce the other identified sources of overhead is provided.

A scheme to implement shared memory with the current passive network interface has been proposed in [KuS85a]. In this scheme, a part of each PE's address space is designated as shared. Assuming a machine-wide shared address space of S bytes, S/N bytes (where N is the number of PEs in the system) of physical shared memory would reside in each PE and each PE would hold memory responding to distinct addresses. An attempt to access a shared memory location for which there was physical memory in the PE would be handled normally. However, an attempt to access a shared memory location which is held by another PE results in a "bus error" trap (due to the non-existent memory) and initiation of exception processing. During the exception processing, the non-local shared address that was generated is examined and the remote PE in which it resides can be determined. A message is sent through the interconnection network to the remote PE requesting the value of the data item at the shared address. The remote PE responds to the request by fetching the data from its local memory and returning it through the interconnection network. As part of the exception processing, the value returned is patched into the run-time stack and the faulted bus cycle is re-run. These actions are equivalent to those that occur in a virtual memory system when an address is found to be non-resident. The performance penalty is rather severe since exception processing is done at both the local and remote PEs and the interconnection network is traversed twice.

It was originally intended to include a DMA controller in every PE. However, implementation constraints lead to the exclusion of a DMA controller so that all data movement to and from the network must be performed by the CPU.

The level of hardware support as provided by the current interface thus does not seem adequate to efficiently utilize the network.

5.4 Network Interface Unit Design

One way to improve message transfer efficiency is to use a dedicated intelligent interface unit, the Network Interface Unit. The use of an NIU for PASM has first been proposed in [TuA83] and [KuS83]. These proposals emphasize the use of the NIU as a sophisticated DMA controller that can transfer nonconsecutive data elements. They did not analyze its benefit for other areas, and they did not propose a specific hardware structure.

The use of communication co-processors in parallel processing systems has precedents. For example, the BBN Butterfly [CrG85] uses a bit-slice microprocessor to monitor every bus access of the Motorola MC68000 main CPU and to route memory accesses either to the local or to remote memories; the Cm* system [SwF77, SwB77, JoC77] uses a co-processor called Kmap to handle global shared memory accesses. The IBM RP3 [PfB85, BrM85] uses a sophisticated cache to route memory accesses either to the local memory or through the network to a remote memory. The Butterfly and Cm* do not provide direct processor-to-processor communication paths; all messages exchanged between processors pass through shared memory. RP3's processors also communicate only through shared memory accesses, but such accesses can cause an interrupt at the destination processor, facilitating message passing. In all of these processors, however, the network is used to pass *shared-memory accesses,* and message passing is a secondary function. In PASM, on the other hand, the network is used primarily to pass messages, and shared-memory accesses are secondary as discuused previously.

Thus, an NIU for the PASM system will have a structure quite different from units in the abovementioned systems, and it will perform different tasks to remedy the overheads of its interconnection structure. In this section, the structure for an NIU will be proposed, and its impact on the message transfer overhead will be discussed.

5.4.1 Structure of the NIU

The components of the NIU are shown in the simplified block diagram in Figure 5-1.

The main CPU of the PASM prototype is an MC68010, and for the full system an MC68020 is proposed. The CPUs are very powerful, but their

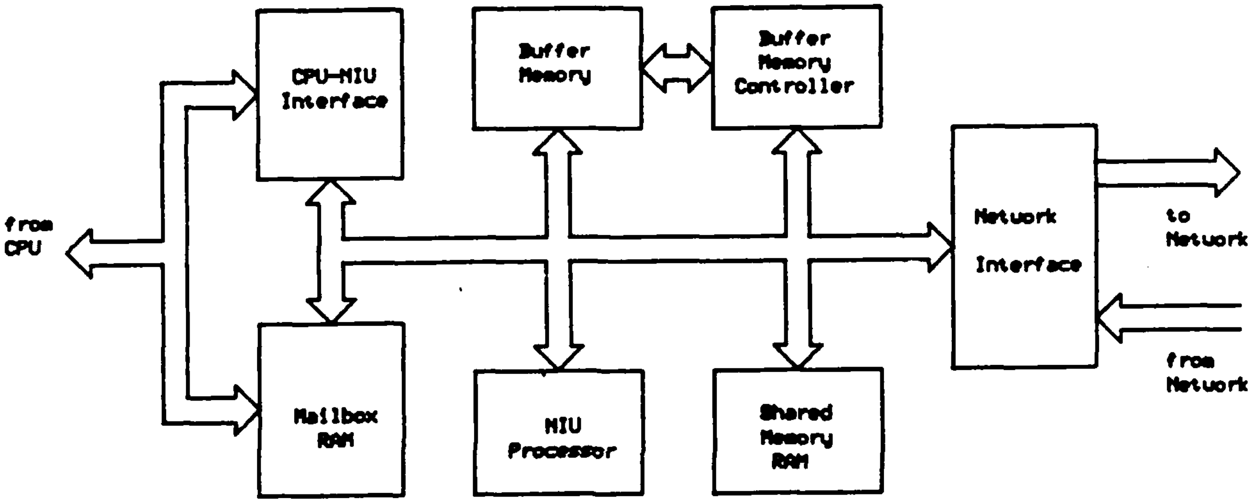


Figure 5-1. Simplified block diagram of the Network Interface Unit.

instruction execution time is in the order of several hundred nanoseconds. The NIU processor is a simple but fast bipolar microprocessor with an instruction execution time of 100ns. In both prototype and full system, this processor can not only work in parallel with the main CPU, but can also execute several instructions for each main CPU instruction. Therefore it can execute simple operations (e.g., address calculations) much faster than the main CPU and reduces the time needed for such operations. The *CPU-NIU Interface* allows the NIU to access its PE's main memory directly. Whenever the NIU needs data from main memory, it steals main CPU cycles and fetches the data. In conjunction with fast address calculations, this facilitates rapid movement of memory blocks into the NIU. The *Mailbox RAM* is a dual-ported memory for passing of control information between main CPU and NIU. If the main CPU needs to send a message, it informs the NIU about the message by filling one of the mailbox slots. The *Buffer Memory* provides memory space to buffer incoming and outgoing messages. The *Buffer Memory Controller* can move consecutive blocks of data in the buffer memory rapidly into or out of the network. The *Shared Memory* provides a slice of the systemwide shared memory. Since the shared memory is not part of the CPU's main memory, the main CPU need not be interrupted when an access to the shared memory has to be performed. The *Network Interface* is the actual port into and out of the network. It has hardware support to automatically generate and check two parity bits for each data word.

5.4.2 NIU Functions

In this section, the functions of the NIU will be examined. The message passing capabilities will be explored first, followed by a discussion on shared memory. A byproduct of the NIU approach, a sophisticated DMA controller, will also be described.

PE-to-PE Message Passing

PE-to-PE message passing is performed under control of the PE main CPU. When a PE needs to send a message to another PE, its CPU places a message transfer request into the Mailbox RAM, and the NIU reads this request. The actual message is stored in main memory, and the NIU can access it there through the CPU-NIU Interface. This message contains the routing and broadcast 'ags, and either the data to be transferred themselves, or a description where to find them.

Obviously a new type of overhead is introduced since the main CPU needs to place the message identifier in memory and has to write the message request to the NIU. However, once the CPU has done this, it is free to continue with other processing, and the only further overhead is the cycle-stealing of the NIU.

While the main CPU performs other tasks, the NIU fetches the message itself from main memory and sets up a path through the network. The impact of the overhead to set up a path is greatly reduced compared to the passive network. The processor is faster, and will therefore perform the necessary operations in less time. It is dedicated, and thus a busy wait does not affect machine performance.

At the destination of a message, the NIU accepts the message and informs the processor that a message is available. The CPU provides the NIU with a destination for the message, and the NIU stores the message directly into the specified memory locations.

The NIU facilitates complex message schemes. One way to specify a message is to give the NIU the location of a buffer in memory, and the size of that buffer, and the NIU will transfer all elements in the buffer to the destination. Especially in image processing, such a simple scheme is not always sufficient. Thus a more complex message facility is provided. A message can be specified by a start address, size, stride, and count. The NIU fetches a data item of the specified size from the start address, and then skips as much memory as the stride specifies, gets another data item of the specified size, and repeats this procedure until all data items as specified by the count have been transferred.

For example, if an 128-by-128 pixel image with one byte per pixel is stored in row-fashion (i.e., the row image pixels are stored consecutively in memory), all elements of column 0 can be transferred by using the first column element as the start address, a size of one, a stride of 128, and a count of 128.

While the use of a slave CPU reduces the overhead for the main CPU, the issue of network under-utilization has not yet been addressed. The buffer memory and the buffer memory controller serve this purpose. A message is not fetched from memory and passed directly to the network but is first stored in a consecutive block in the buffer memory. During this time, the path is not yet established. In this way the network path is not kept open during the time needed for address calculations. After the block is available in the buffer memory, the path is established, and the buffer memory controller uses high-speed hardware to move the block into the network at the networks maximum speed. At the destination, the buffer memory controller accepts the block at this high speed. When the block has been sent, the path is dropped. Thus the network is used only for the shortest possible time, and network conflicts will be reduced.

Data Transfer in SIMD Mode

As shown in the analysis of the routing tag calculation overhead, a simple correspondence between logical and physical PEs exists. The NIU takes advantage of this fact in the following way. In the PE address space, a range of N addresses is reserved, beginning at location B. Whenever the CPU writes a data word to location B+i, the NIU interface detects the access, stores the data word and the offset i, and informs the NIU processor of the pending access. The NIU uses three data words identifying partition and fault location (that are maintained by the operating system) and the offset i to transfer the data item to the logical PE i of the current partition. It first verifies that a valid destination is specified by comparing i with the partition size. If i is larger than the partition size, the NIU terminates the write cycle through a bus error, indicating to the PE CPU that an error occurred. If i was permissible, the NIU calculates the routing tag and transfers the data item to the destination. At the destination, the NIU CPU writes the data word it received into a reserved location of the Mailbox RAM. Since PEs are always synchronized, the destination PE CPU need not be notified that a data word is available. This method facilitates SIMD data transfers without the need of operating system calls, thus eliminating a major source of communication overhead.

Shared Memory

As was previously discussed, shared memory is desirable in the PASM system, and it can be emulated using no NIU, though with a severe performance penalty. Through the use of an NIU, this penalty is significantly reduced. Since shared memory accesses are costly even with an NIU, not all memory in the system is shared to reduce the danger of overusing shared memory (e.g., running a program that resides in another PEs shared memory space would be extremely slow). Instead, every NIU in the system contains a slice of the globally shared memory. For the prototype, an NIU will contain 2K byte of shared memory, resulting in a total shared memory size of 16*2K byte = 32 K byte. Whenever a processor accesses any location in the shared memory space, the access is decoded by the local NIU. If the access was directed to the shared memory slice located in the local NIU, the NIU performs the access on the memory without having to use the interconnection network. If the access was directed to any other shared memory slice, the NIU determines the PE in which the slice is located, sets up a path to the proper remote PE, and send a message to the remote PE requesting shared memory access. This message is handled not by the remote CPU but exclusively by the remote NIU. The remote NIU accepts the message and performs an access to its shared memory slice. The source drops the network path as soon as the complete message has been passed to the destination. If the access was a write, the remote NIU writes the data item into its shared space, and the access is complete. If it was a read access, the remote NIU sets up a path to the source PE, and passes the result of the read operation to it. Using this scheme, processing in the remote CPU is not interrupted, and the fast NIU processor results in high speed accesses.

One problem that occurs is that the remote NIU has to establish a path to the source of the request if a read access is to be performed. This has several disadvantages. The source address has to be passed as part of the message which requires network transfer time; the destination NIU must set up a path through the network which not only takes time but there no guarantee that such a path can be established at all; and read-modify-write cycles are not possible due to deadlock. A bidirectional network solves these problems. Write accesses are performed as described above. During a read access, after the destination address has been sent, the data direction of the network is reversed, and the data flows from destination to source. Thus a shared memory access is guaranteed to complete once a path has been set up from source to destination. No deadlock can occur during read-modify-write cycles. After the network path has been established, a read operation can be performed, followed by a write operation.

DMA Controller

A byproduct of the NIU is its capability to function as a DMA controller. As mentioned earlier, the NIU can not only fetch data items that are stored consecutively but also data that is stored periodically by defining size, stride, start address, and count. These four parameters can be defined for a source and a destination in the PEs main memory, resulting in a powerful memory-to-memory DMA capability that is very useful for reformatting. A simple example is the rearranging of an image from column to row storage format. A sophisticated compiler could use this capability to move arrays or more complex data structures.

5.5 Summary and Future Research

In this chapter, the overhead incurred in a PE-to-PE interconnection network has been analyzed. A Network Interface Unit that greatly reduces the overhead has been introduced, and some of its possible functions have been described. An NIU for the PASM prototype is currently being designed. While the analysis in this chapter proved that an NIU is advantageous, it did not quantify the performance gain. Further research will provide this quantification by analyzing in detail the current implementation of a passive network interface and the software overhead involved, and by analyzing the performance of the NIU implementation. This analysis will show which communication functions are best performed by the main processor, and which should be performed by the NIU.

CHAPTER 6

DESIGN OF A 1024-PE PASM SYSTEM

An approach for implementing a PASM system with 1,024 PEs in the computational engine is in Appendix B.

CHAPTER 7

INTELLIGENT OPERATING SYSTEM CONCEPTS

7.1 Introduction

A system for executing image understanding tasks on a reconfigurable parallel architecture like PASM has been proposed in [DeS85]. Figure 7-1 shows a system model. An *Intelligent Image Understanding System* accepts input from image sensors and produces as output a scene description. It uses a *knowledge base* to determine which subtasks have to be performed to complete the image understanding task, and it uses an *algorithm database* to find the appropriate algorithms for the subtasks. It informs the *Intelligent Operating System* about the subtasks to be performed and about the precedence of these subtasks. The Intelligent Operating System selects parallel implementations of the subtasks using its own algorithm data base. It uses its knowledge base to schedule the subtasks, and configure PASM to achieve optimum performance. It takes other currently executing tasks, task priority, and the current availability of resources into account to reach a decision. The Intelligent Image Understanding System and the Intelligent Operating System may employ expert systems to arrive at intelligent decisions. The *Low Level Operating System* routines interact with the hardware and perform the actual reconfiguration, and the *Reconfigurable Parallel Processing System,* i.e., PASM, performs the actual computations.

The decisions of the Intelligent Image Understanding System are guided by the images being processed (e.g., their noisyness, content, etc). The knowledge

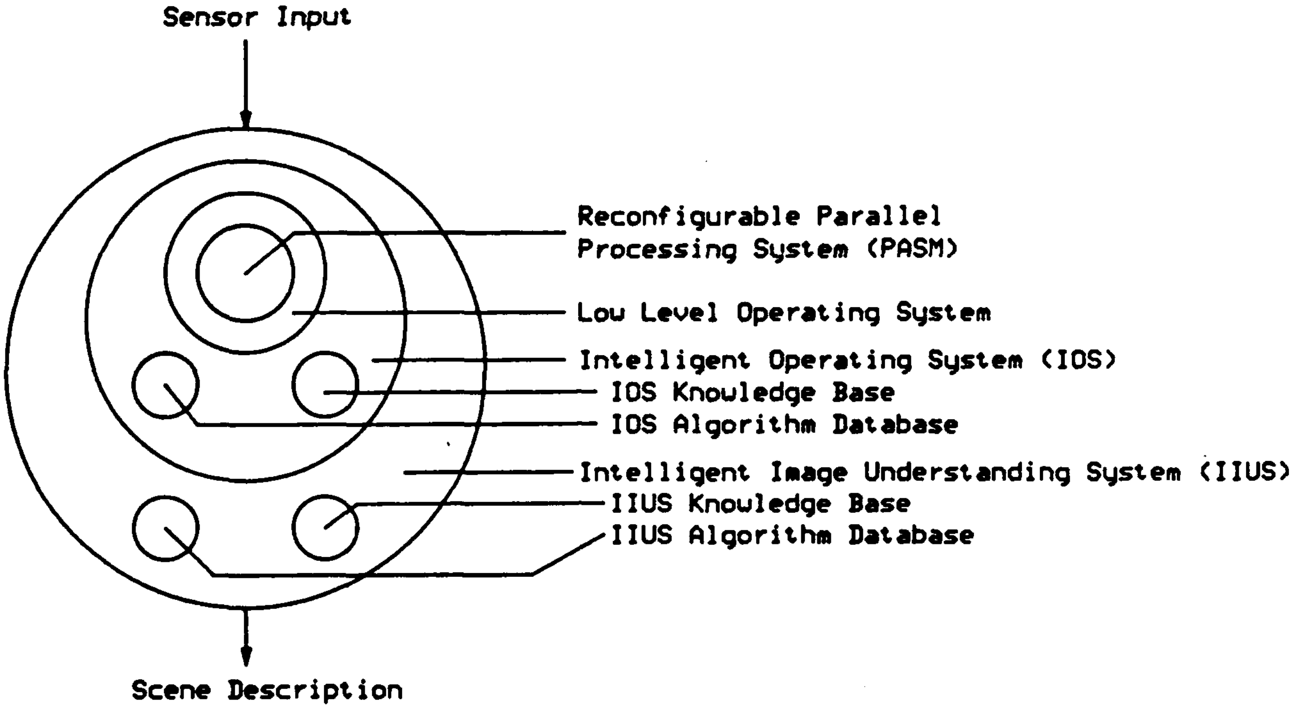

Sensor Input
Reconfigurable Parallel Processing System (PASM)
Lou Level Operating System
Intelligent Operating System (IOS)
IOS Knouledge Base
IOS Algorithm Database
Intelligent Image Understanding System (IIUS)
IIUS Knouledge Base
IIUS Algorithm Database
Scene Description

Figure 7-1. System Model.

required for its decisions are therefore the concern of image understanding research and are not part of this work. The Intelligent Operating System is given by the Intelligent Image Understanding System a set of subtasks which it has to execute.

One way to provide this set is through a precedence graph that specifies the subtasks are to be run and their relative order. In addition, the Intelligent Operating System may be implemented using parallelism, and then must itself be considered as another subtask competing for resources. The Intelligent Operating System must schedule the subtasks such that a minimum overall execution time is achieved. This problem is complicated by the fact that PASM is a dynamically reconfigurable architecture. Thus subtasks can be executed in different modes of parallelism, and the number of resources assigned to each subtask can be varied. Since the run time of an algorithm usually is not a linear function of the number of resource assigned to it, the scheduling problem is not straightforward. In many cases, the run time of an algorithm is data dependent, and only estimates may be available. These estimates could be updated at runtime, and the Intelligent Operating System has to adapt the schedule, possibly reconfiguring the system. In reaching the decision whether to reconfigure or not the Intelligent Operating system must take the overhead incurred by reconfiguration into account.

Furthermore, results from the execution of subtasks will cause changes in the precedence graph so that the system needs to adapt to changing requirements at runtime. For example, if it is detected that a part of an an image contains an important feature, it is desirable to concentrate computing power on that image part to further analyze that feature. It is also desirable to concentrate computing power on the execution of subtasks that have been determined to be important at runtime, and thus some measure of priority might be associated with each subtask and must be accounted for in the scheduling algorithm. Concentrating of computing power on part of an image or a subtask at runtime requires the change of resource allocation, dynamic updating of the current subtask schedule, and movement of programs and/or data between processors. The scheduling algorithm must be able to respond to such an environment, and it must be able to decide which reconfigurations have to be executed to achieve optimum results. The Intelligent Operating System thus needs knowledge about PASM and about the Low Level Operating System to schedule the subtasks and reconfigure PASM. It must know which algorithms exist to perform each subtask, needs the run time as a function of assigned resources, must know which possibilities to reconfigure a given program exist, what the parameters of this reconfiguration are, etc. The goal of this research is to determine and specify the procedures, techniques, and knowledge required to design such an Intelligent Operating System. knowledge. The information can then be incorporated into a knowledge base and rule set for an expert system, or it can be used in some other implementation of the Intelligent Operating System. Thus, the goal of this research is not the actual implementation of the Intelligent Operating System, but theoretical research that will make it possible to have the Intelligent Operating System make intelligent and reasonable decisions at execution time in a dynamically reconfigurable environment. The results of the work will also be of interest and applicable to many other current and future parallel processing systems.

The steps leading to the acquisition or the required knowledge will now be outlined. The first two steps are discussed in detail in this thesis proposal, the others are topic of future research.

[1] **Modeling of PASM.** A three-level model of the PASM system is basis for all further analysis. The model are the Hardware Level Model, the Interrupt Level Model, and the Reconfiguration Level Model.

[2] **State of PASM.** The three levels of models are used to analyze the PASM state in three levels. This state will be used to make statements about scheduling, configuration, and reconfiguration of PASM.

[3] **Job Scheduling in PASM.** Previous work on scheduling in PASM will be analyzed, and alternatives will be developed. The job scheduling procedure and the job scheduling time parameters will be analyzed.

[4] **Analysis of job scheduling utiiizing precedence constraints.** The Intelligent Image Understanding System provides the Intelligent Operating System with a set of tasks which have to be executed under precedence constraints. The Intelligent Operating System can select the resources (i.e., PEs) it wants to assign to each task, and it can find the run time of an algorithm as function of the number of PEs in the algorithm data base. The problem of finding an optimum schedule given precedence constraints, the algorithm run time as a function of the assigned processors, and PASM parameters will be examined.

[5] **Reconfiguration in PASM.** Ways to reconfigure jobs in PASM will be analyzed. Three different alternatives are reconfiguration using no multiprogramming, limited multiprogramming, and full multiprogramming.

[6] **Low Level Operating System Interface.** When the methods of configuring and reconfiguring PASM have been analyzed, low-level operating system routines can be defined that supply parameters about PASM and perform the actual reconfiguration.

7.2 PASM modeling

The *model* of a system is a description of all time-invariant elements in the system. The emphasis of a model is placed on *time-invariant*. It is the basis of all further analysis, since it shows all elements in the system, describes their purpose and their function, and indicates time-variant elements.

As mentioned above, three levels of models will be discussed: the *Hardware Level Model,* the *Interrupt Level Model,* and the *Reconfiguration Level Model.* The Hardware Level Model is closely related to the physical machine, and it is basically a complete description of all components and functions of the machine. It provides the hardware engineer with a basis for analyzing and testing the system. A system programmer writing an operating system could extract all knowledge required for the operating system from the Hardware Level Model, but this task would be difficult since much of the information provided in the Hardware Level Model is irrelevant for the programmer. To avoid overloading the system programmer with details, a second model, the Interrupt Level Model, is introduced. It contains all information required to write low-level operating system routines. These routines deal with individual processors and communication between processors, but do not consider the system as a parallel machine with special capabilities. The Reconfiguration Level Model provides the knowledge to program the Intelligent Operating System. It shows the possible partitions of the system, the MC group structure, and a simplified PE model. The three levels of models will now be discussed in detail.

7.2.1 Hardware Level Model

The Hardware Level Model consists of the schematic diagrams (which show all interconnections of individual components), timing diagrams (which show the time relationship between signals in the system), the data sheets of all individual components (which describe the electrical and logical behavior of the components), and the description of the mechanical construction of the system (which allows the hardware engineer to locate logical signals on physical boards). By using the Hardware Level Model, the behavior of the machine can be predicted; this prediction and a comparison with measured signals makes it possible to verify correct operation and to locate and correct hardware faults.

One important aspect of the Hardware Level Model is that it deals with physical components. As an example, consider the block diagram of a PE as shown in Figure 7-2. The blocks shown represent physical boards, and thus two memory boards are shown even though they might provide a continuous address space and thus be a single logical memory. The Instruction Broadcast Interface and PE/MC data and Control Interface, on the other hand, are placed in a single block even though they represents two functional units.

The discussion of the individual boards of the PASM prototype in Chapter 4 is the kind of information the Hardware Level Model provides. As will be seen in the next section, much of the detail shown in those description (which are not even the complete hardware descriptions) will not be required in the Interrupt Level Model.

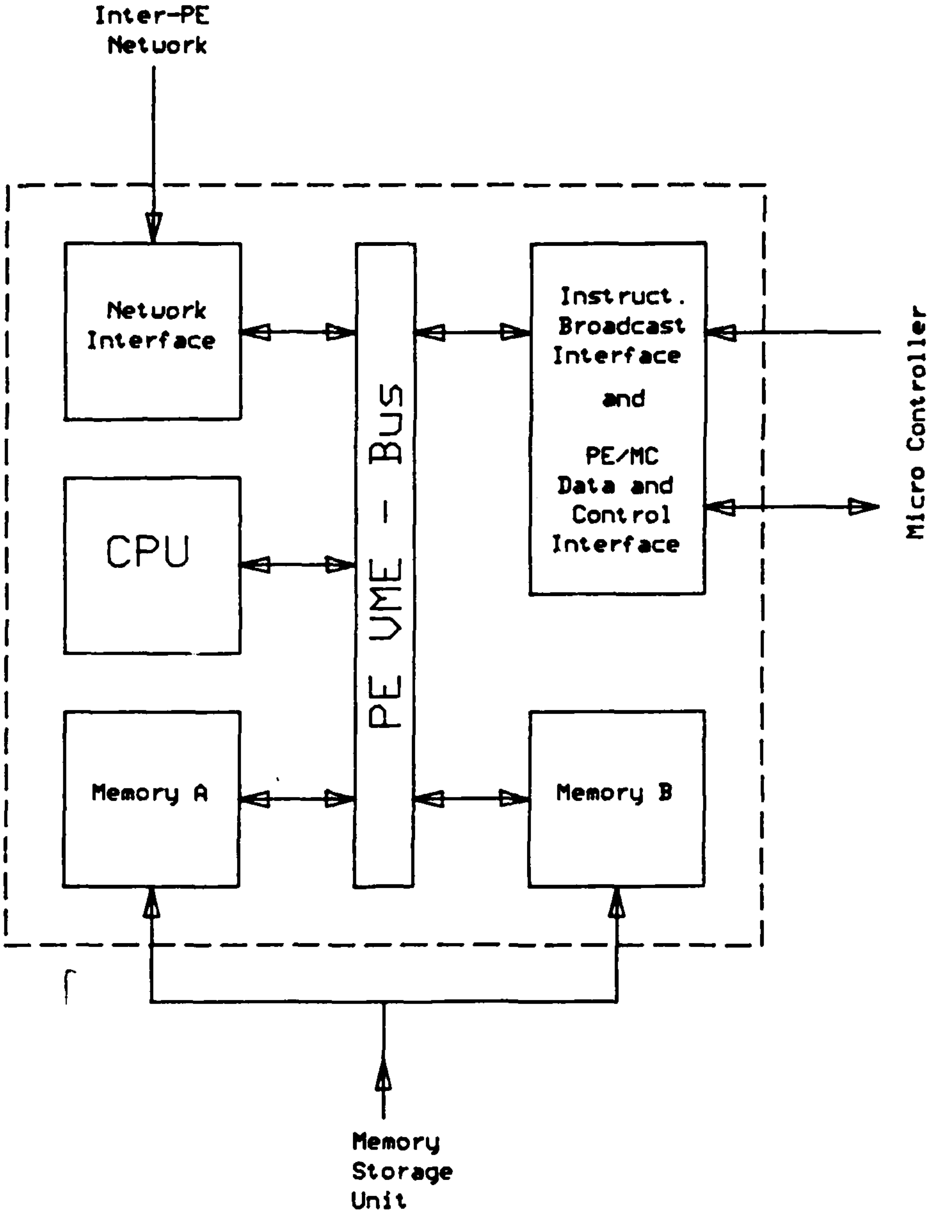


Figure 7-2. Hardware Level Model block diagram of a PASM PE.

7.2.2 Interrupt Level Model

The operating system of any computer shields the hardware from the user by providing high level services like file service, I/O, and process control. Thus user programs usually are not concerned with the actual hardware structure of the machine, especially when written in a high level language. However, when programming the operating system, the knowledge of much of the hardware is essential for efficient operating system function. The Interrupt Level Model provides the system programmer with all the hardware knowledge he needs to write low level operating system routines, while avoiding to overload him with the detail provided by the hardware model. Low level operating system routines include memory management, low level I/O, interrupt handlers, device drivers, and context switching. Contrary to the hardware engineer, the system programmer is not interested in the individual lines or timing relationships. He deals with processor behavior, memory size and memory locations, virtual memory, I/O devices and their addresses, interrupts, etc. Depending on the task the system programmer needs to perform, more or less detailed knowledge of the hardware is required.

Even though PASM is a multiprocessor system requiring special operating system functions to run parallel programs, it consists of a number of individual processors, each of which needs a small individual operating system. The parallel operating system will use the individual operating systems, simplifying the overall design. Therefore one part of the Interrupt Level Model is the structure of each processor in the system, with a description of the individual components as required by the system programmer. The other part of the Interrupt Level Model shows the inter-processor communication channels so that the system routines dealing with processor interactions can be written.

Individual Processor Model

Consider the Interrupt Level Model of a processor as shown in Figure 7-3. The CPU block is connected to the system bus through the Memory Management Unit (MMU). CPU and MMU need not necessarily be separate units. Many modern microprocessors like the Intel iAPX286 and the Zilog Z80000 have on-chip MMUs. The Motorola MC68020, which is proposed for use in a full PASM system, needs a separate MMU chip. The system bus is then connected to m I/O modules and n memory modules. Depending on the CPU type, I/O devices may appear as memory locations (memory mapped I/O, e.g. Motorola MC68000 family processors), or they may appear in a special I/O space (e.g., Intel 80286). Sometimes it may not be clear whether a device is memory or I/O. An example is shared memory that is used for message passing. In such a case, the type will be determined by definition. Each of the modules in this block diagram requires an *appropriate description* so that the system routines can be written.

CPU block. One important information for the programmer is the type of CPU, and with that the CPU programmer's model which contains the instruction set, register model, exception handling, etc. The details can be retrieved from the CPU programmer's manuals. Most of the support logic in the CPU block like bus drivers, address decoding, interrupt logic, etc., is invisible to the software and need thus not be included in the model.

Memory management. The memory of a processor is a resource that needs to be allocated and deallocated to user programs running on the machine. The memory management routines must make sure that user programs do not overrun each others memory space, they must allocate and deallocate buffer space, and they must prohibit memory overflow. If the memory is

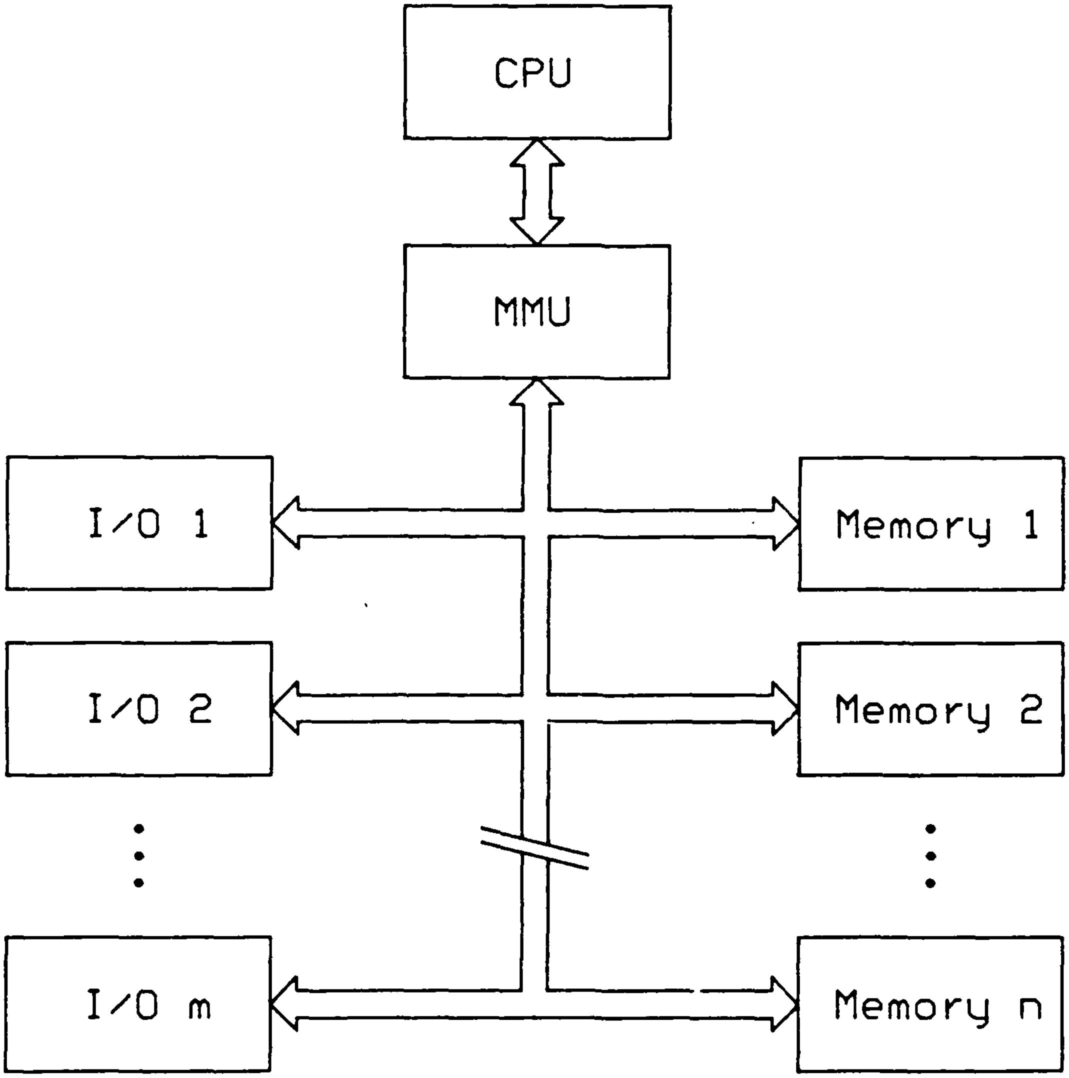


Figure 7-3. Interrupt Level Model of a processor.

demand paged, the routines must also perform the virtual memory administration. The Memory Management block shown in Figure 7-3 - if included in the processor - supports these operating system tasks in hardware. It translates physical to logical addresses and performs checks on the validity of the memory access. Many different types of memory management are possible, and their modeling will differ from implementation to implementation. If a single-chip memory mangement unit is used (for example, the prototype uses MC68451 MMUs [Mot83]), the model will be the programmers model as described in the data book. Some CPUs (e.g., Intel's iAPX 286) have the memory management system already built in. Here the CPU data book will also yield the desired MMU model. If a custom memory management unit is used, an appropriate model must be provided by the hardware designer. The memory management model gives the system programmer the knowledge to program the MMU. He will know which registers to load, how to protect memory space, how a protection violation is flagged, how to perform logical to physical address location, etc.

Memory Units. The following parameters are required to to describe a memory module:

Memory type. Memory can be read/write memory, or it can be read only memory.

Memory access protection. Memory can not only be protected from unauthorized access by the MMU, but hardware protection is possible as well. This might result in memory that can only be accessed in a supervisory state to provide memory space for operating system tables.

Memory start and end address. The programmer needs the physical address space in order to map the logical space into it.

Memory access time. Some frequently used routines should run as fast as possible. It is important for the system programmer to place such routines into memory space with the lowest available access time.

Data path width. Some systems (e.g., Motorola's MC68020) allow various data path widths of the memory. The data path width influences performance and is thus an important piece of information for the system programmer.

Data integrity protection mechanism. In order to ensure data integrity against soft and hard memory errors, many memories have either parity protection (which detects all single bit faults) or Error Detection and Correction (EDC) circuitry (which detects all double bit errors and corrects all single bit errors). The system programmer needs to know about these mechanisms to react properly if a protection violation is indicated and to avoid using unprotected memory areas for important data. If EDC is available, periodic *scrubbing* should be performed, i.e., all memory locations should be read from time to time, thus correcting all single bit errors that have occurred since the last access.

Shared/not shared. Shared memory areas pose special problems of data consistency. The programmer needs to know about the sharing of memory to take any precaution that might be necessary.

I/O modules. Since all I/O modules will perform different kinds of operations, one important part of the model is a functional description. This description will greatly differ from the functional description given in the hardware model, since other parameters are relevant for the system programmer than for the hardware engineer. It will often be the case that the functional description is not stand-alone, and other parts of the system (i.e, corresponding I/O modules on other processors) may have to be studied in order to understand the function of the I/O device in question. In the functional description, the registers of the I/O module, their functions, address, and interrupt capabilities are explained. As an example, the Interrupt Level Model of the Parallel Port Board will be discussed here. The model is contrasted to the hardware description of the same board that is given in Chapter 4.

The Interrupt Level Model block diagram of the Parallel Port Board is shown in Figure 7-4. Only one of the parallel ports shown in the hardware description is included here since the two ports are lcgically equivalent. The VMEbus Interface is omitted as well since it is invisible to the programmer. The parallel port itself is shown in greater detail than in the hardware description. This does not imply that the hardware description does not provide this information; it is just not included in the simplifying block diagram but can be found in the more detailed descriptions. The system programmer, however, deals with individual registers as shown in the diagram.

The parallel port chip MC68230 can be accessed at base address $FE6180. The programmer's model and thus the offset for the individual registers can be obtained from the MC68230 data sheet. Port A is connected to the lower eight data bits D0-7, and it can cause an autovectored interrupt on interrupt level 3. The output of Port A is interfaced to the outside world through a bidirectional driver, and the data direction is determined by setting Port C bit PC0. PC0 = 0 sets the driver to output, PC0 = 1 sets it to input. The system programmer must take great care that the data transfer direction of the parallel port is identical to that of the drivers; otherwise hardware failure can result. Port B is identical to Port A, except that the driver direction is determined by PC1.

The parallel port also provides a timer which can interrupt the CPU after a software-selectable time by an autovectored interrupt on level 2. Control

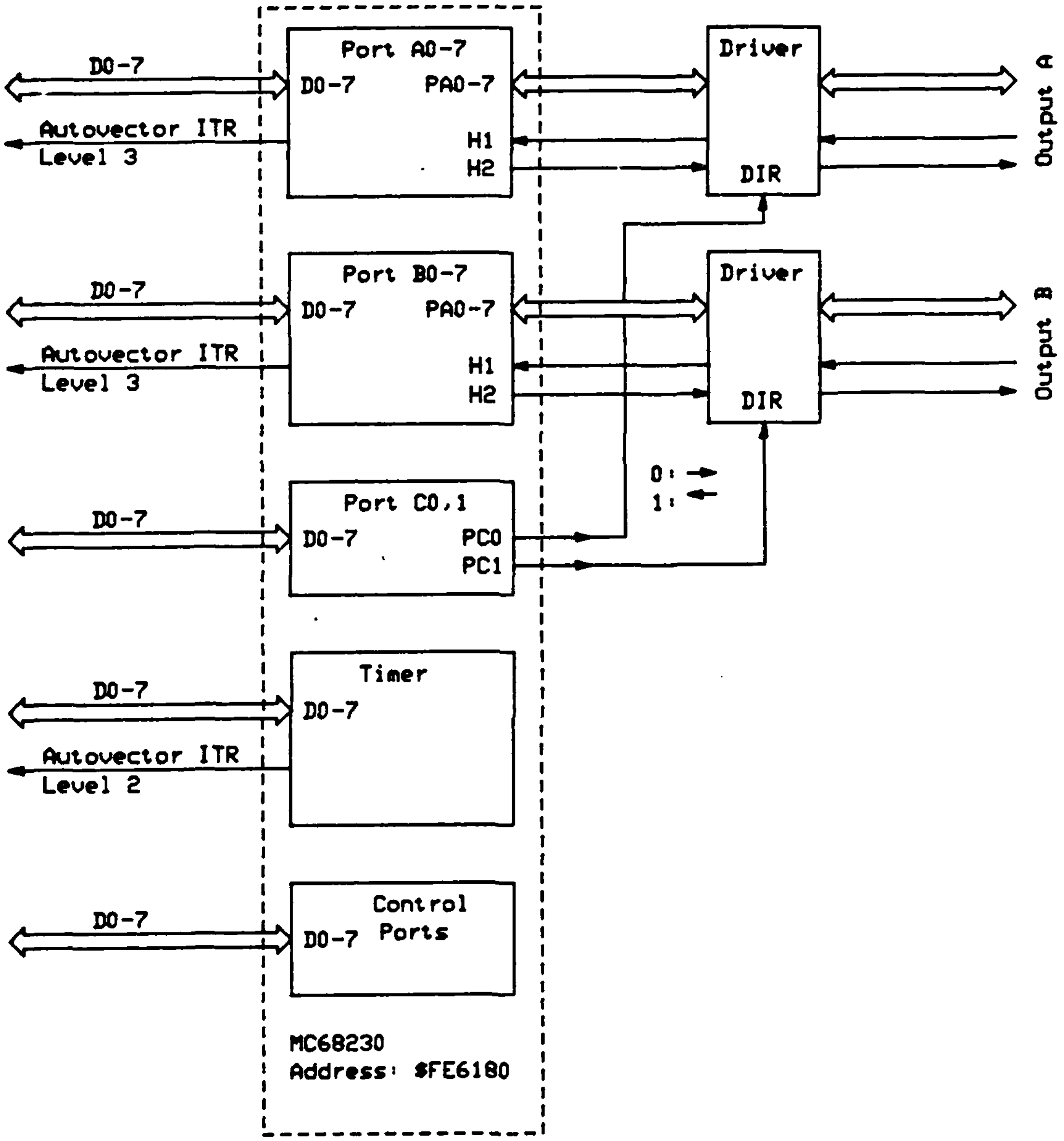


Figure 7-4. Interrupt Level Model of the Parallel Port Board.

ports in the MC68230 are required to set up the output ports, the timer, and the interrupt lines.

As mentioned previously, knowledge of other parts in the system may be required for proper programming. In the case of the parallel port, the data direction of the port must be compatible with the data direction of the port it is connected to. For example, if Port A of this module is connected to a port B of another module, and Port B is set to output, then Port A must be set to input.

Processor Interactions

As has already been alluded to in the discussion of I/O modules that communication channels between the individual processors have to be known to the system programmer. Figure 7-5 shows the Interrupt Level Model of the the communication channels in the PASM prototype. The diagram would be similar for a 1,024 processor PASM; just more MCs and PEs would exist. Four different kinds of channels are shown. The parallel port connections are used to exchange control information between various processors in the system. The MCs exchange information with the PEs and send SIMD instructions to the PEs through the MC/PE Channel. Bulk data transfer (e.g., moving data from disk to memory and vice versa) is handled through DMA channels which directly access the memory of their destination through a private port. The PEs communicate with each other through the Inter-PE Network which is not shown explicitly. Serial port connections facilitate communication of the prototype with the Engineering Computer Network.

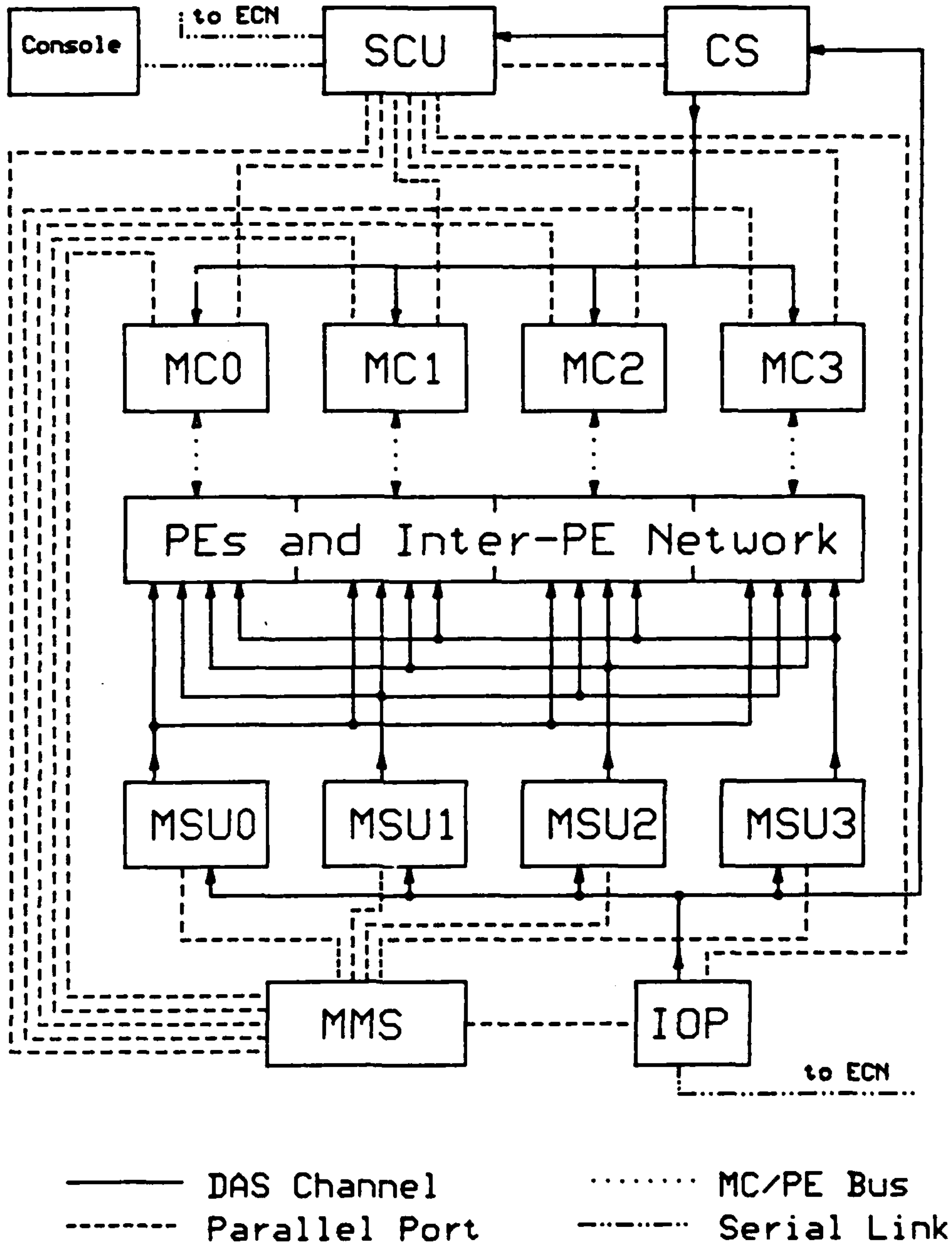


Figure 7-5. PASM Communication Channel Model

7.2.3 Reconfiguration Level Model

With the help of the Interrupt Level Model the system programmer can write all operating system programs that handle basic operations. Some of these routines consider the parallel nature of the PASM system, but only to a limited extend. The Intelligent Operating System of PASM must handle these parallel aspects. PASM is a dynamically reconfigurable SIMD/MIMD machine, and the operating system must support the reconfiguration capabilities. One reconfiguration is the choice of SIMD and/or MIMD mode. This choice will be made by the user, either directly if using an explicit parallel programming language, or implicit through the use of a parallelizing compiler. In the same way the user will also specify the extent of parallelism of the program, i.e., he will specify how many PEs the program needs. The operating system must schedule the appropriate number of PEs, load the required data and programs, and supervise job execution.

The first part of the model thus has to show possible PE allocations. Figure 7-6 gives a system model. The SCU controls a set of Q MC groups, and each MC group contains a number of processors (four in the prototype, 32 in the 1,024 PE system). The processors contained in an MC group are the smallest number of processors that can be assigned to a job. Even if a program needs only a single PE, it will still be assigned a complete MC group. On the other hand, multiple MC groups can be assigned to a single job. The set of MC groups that are combined to work together on a job is called *partition*. Partitions always contain 2^i MC groups, with i ranging from 0 to $q=\log_2 Q$. If R MC groups are combined, their numbers agree in their low-order $q-r$ ($r=\log_2 R$) bit positions. The secondary memory system provides file service to the MC groups and the SCU. Low level operating system routines support this file

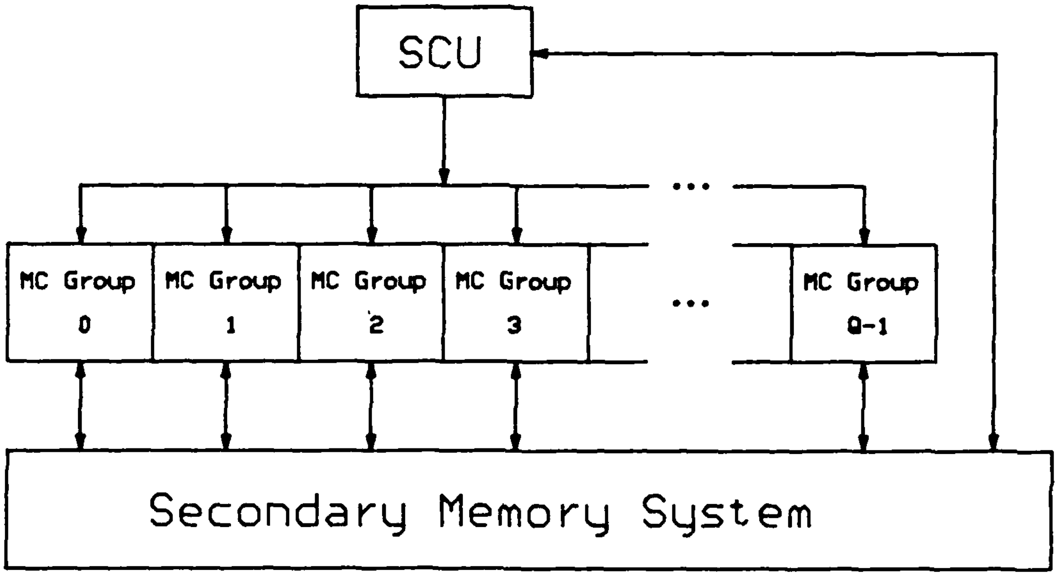


Figure 7-6. Reconfiguration Level Model of PASM

service.

Each partion can run in SIMD mode, in MIMD mode, or a program might exploit both modes. The MC group model as shown in Figure 7-7 shows how these modes are supported. The MC group controller supervises execution of SIMD and MIMD programs. If an SIMD program is required, it schedules the SIMD job on the SIMD controller and directs the PEs to go to SIMD mode and fetch instructions not from their own memory but from the SIMD controller. In MIMD mode, the MC Group Controller informs the PEs where in their memory the program is located, and they start execution individually. Note that a distinct MC Controller and SIMD controller are abstractions. As has been shown in the description of PASM (and in the hardware and interrupt models), each MC group has only one microprocessor associated with it. The MC group Controller and SIMD controller are therefore physically the same processor, but logically they perform different functions.

The Inter-PE network is shown in the diagram, as well as shared memory. This again is an abstraction. The shared memory facility of PASM utilizes the interconnection network. Since the high level system programmer is shielded from this implementation, network and shared memory can be shown as two distinct units. The secondary memory plays a very important part in job scheduling, since data and programs must be moved from secondary storage to memory before execution of a job can begin.

A diagram of the individual PEs is shown in Figure 7-8. It basically ignores all details of I/O etc., but it shows three distinct memory modules. Mem A and Mem B are dual-ported and can be accessed directly from secondary memory, while the local memory is accessible only through the CPU. This has important implications for job scheduling. For example, if a job is

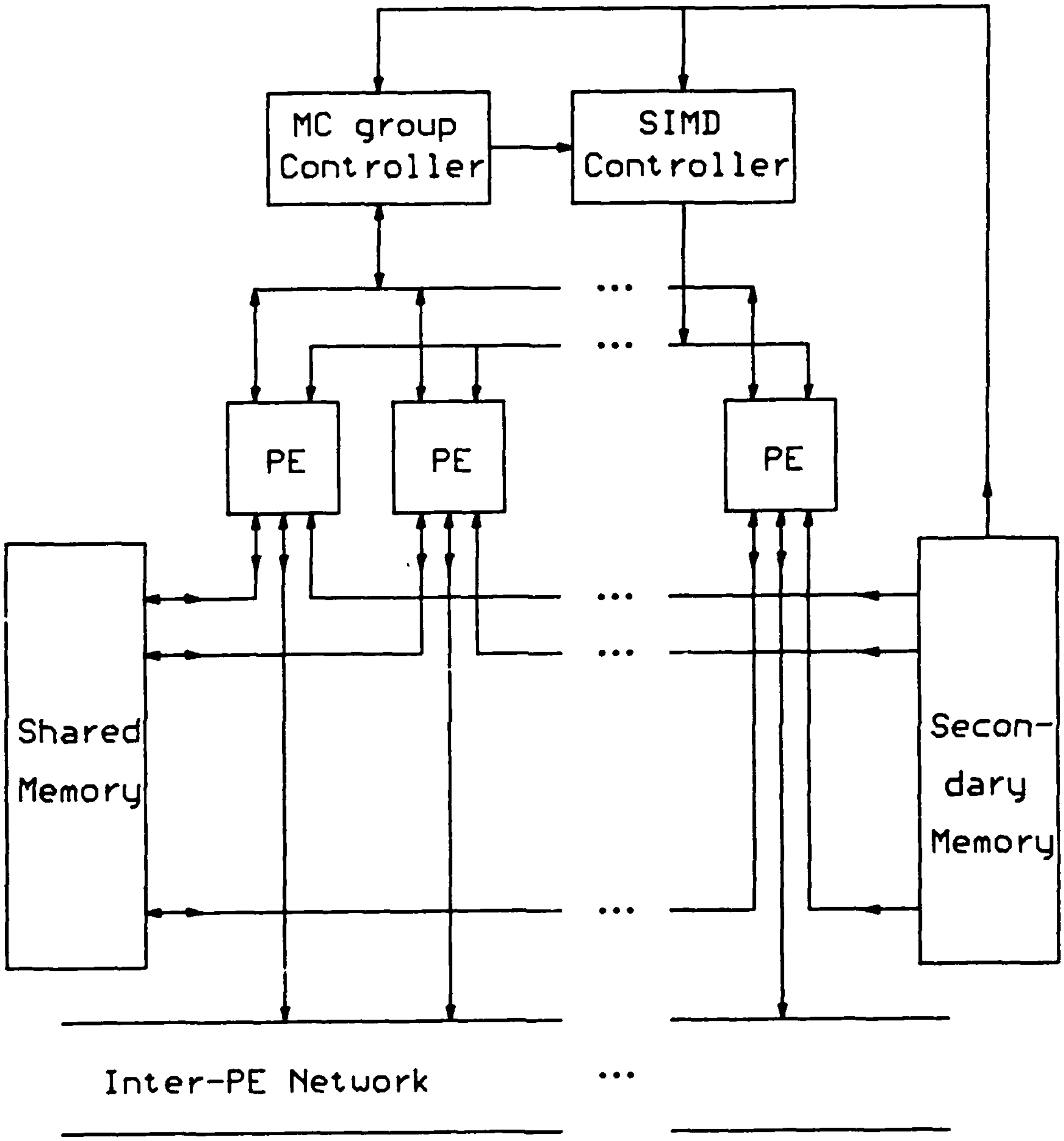


Figure 7-7. Reconfiguration Level Model of an MC group.

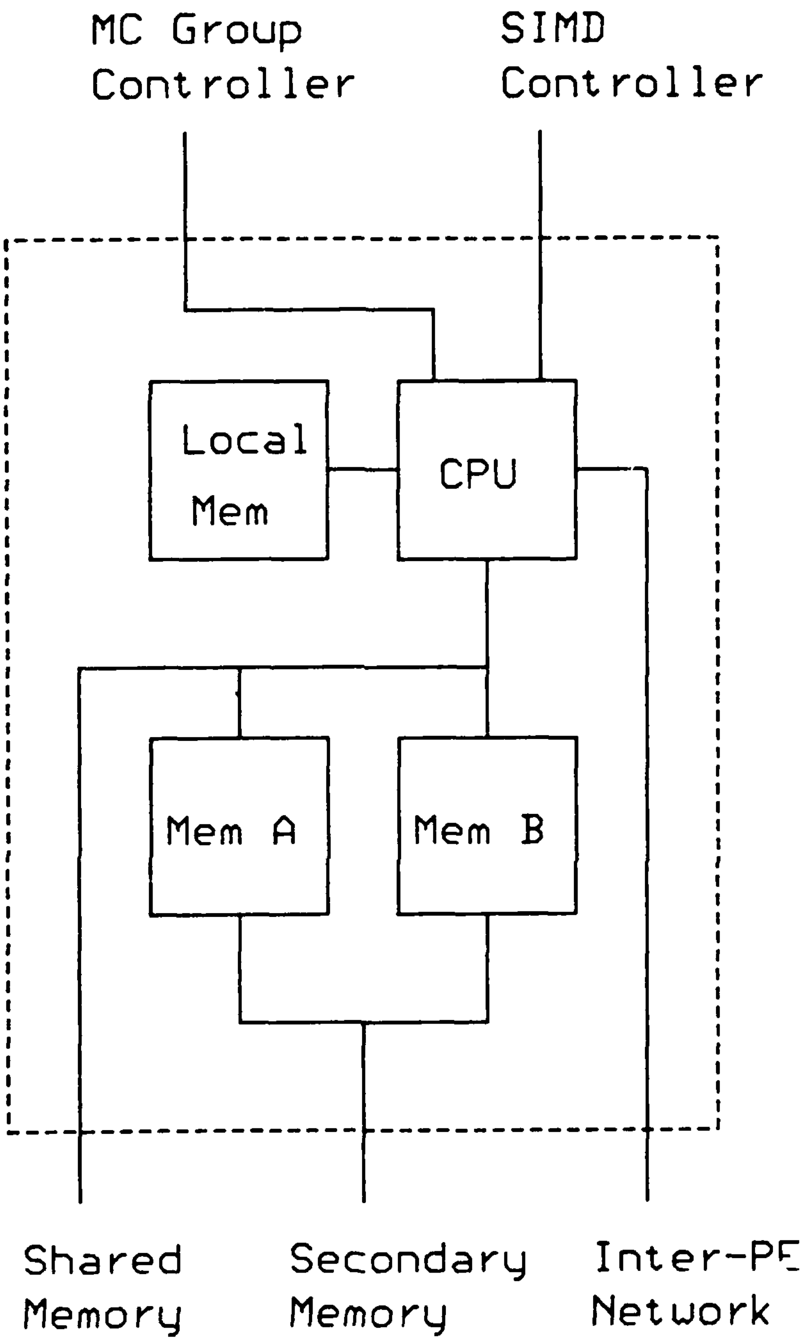


Figure 7-8. Reconfiguration Level Model of a PE.

currently running using only Mem A, the secondary memory system can be instructed to load a new job into Mem B. The load process can thus be accomplished without degrading the currently running job. As soon as this job is completed, the job that was loaded into Mem B can be started without waiting for I/O. Job scheduling based on two memories has been analyzed by Tuomenoksa and Siegel [TuS81, TuS82, TuS84, TuS85].

7.3 The PASM state

In the previous section, models of the PASM system were introduced, and these models describe the static properties of PASM. The models provide the system programmer with the knowledge about the structure of the system. This knowledge is required for initialization, device drivers, etc. When the operating system is running, it needs information about the current *state* of the system to make decisions (e.g., which job to run next, which interrupt to service first). The state of a system is the momentary value of all time-variant variables in the system. The emphasis here is on *time-variant,* in contrast to the static model. No static variable or element is part of the system state since these are already sufficiently described in the system model. For example, the statement *PASM has 1024 PEs* is not part of the state description since it is time invariant. On the other hand, the statement *PASM has 1024 PEs that are fault-free* is part of the state since the number of working PEs changes whenever a fault occurs or is repaired. Note that the model is a necessary prerequisite for the state. For example, the model will be needed to make relative statements like "90% of the memory is used." For such a statement the absolute size of the memory is required which is part of the model.

Corresponding to the three levels of models, three levels of states will be discussed here: the *Hardware Level State,* the *Interrupt Level State,* and the *Reconfiguration Level State.*

7.3.1 Hardware Level State

In the Hardware Level Model, a complete description of the PASM hardware in form of block diagrams. schematics, timing diagrams, data sheets, etc., is given. All signals in the system are identified. The Hardware Level State describes the current condition of components and assigns values to signals.

The value of a signal line can be described in several ways. The electrical potential at a given point in time is the description closest to the physical signal. If a signal is analyzed on an oscilloscope, the electrical value of the signal over time is displayed. No physical to logical mapping occurs, and this precise physical description is very important when debugging hardware. For example, a signal might be in an indeterminate logical state; a logic analyzer would interpret this indeterminate state as either high or low, leading the engineer to erroneous conclusions. An analog analysis will show the actual conditions and will permit detection of errors not detectable by logical analysis.

The logical description is an abstraction of the electrical signal. The electrical value of a TTL signal can be in the range from zero to five volts, and if the value is between certain limits, it will be interpreted as a logical *high* or *low.* If it is not between these limits, the signal state is indeterminate. Other possible states are tri-state (i.e., open connection), rising transition and falling transition. With these states, the logical value of any signal can be described.

The logical values of ***high*** and ***low*** mean ***true*** and ***false*** only if positive logic is used; if negative logic is used, ***high*** means ***false*** and ***low*** means ***true***. In the discussion of signals, a further abstraction is therefore helpful. If a signal is true, it is said to be asserted; if it is false, it is said to be negated. Assertion and negation are independent of the actual electrical and logical value of the signal.

Signals can be grouped for easier description. For example, the state of a bus (e.g., a data bus) is completely given by the value of all bits on the bus. This value can be given in hexadecimal form. Assuming a 16-bit data bus is to be described, the state can be given by the statement "The data bus has the value $3AD9_{16}$." From this the hardware engineer can determine the electrical and logical state of all 16 signals of the data bus.

Externally accessible signals as described above are not the only components of the state. Also relevant are values internal to components. Examples are the values of memory cells and registers, or the state of a finite state machine.

Another part of the Hardware Level State is the state of components. For example, drivers (e.g., data or address drivers) can be enabled or disabled, flip-flops can latch incoming data on positive or negative transitions, etc. Consider as an example two drivers driving the same bus. At any given time, only one of them must be enabled or failure will result. The signals as seen at the output of both chips will be identical, even though one of the chips does not drive the signals. Thus the state of the component may be important.

A last part of the Hardware Level State is the fault condition of a component or interconnection line. It can be fault free, or it can be faulty. Many fault conditions are possible, e.g., stuck-at-zero, stuck-at-one, open connection, etc. A fault condition might not always influence the state of signals. For example, if the output of an AND-gate is stuck-at-zero, and one of the inputs is zero, the fault could not be detected, and thus the signal state alone is not sufficient to describe faults.

7.3.2 The Interrupt Level State

While the Hardware Level State is important for the hardware enginer to debug hardware and verify correct operation, the Interrupt Level and Reconfiguration Level States are relevant for system software programmers and users. Similar to the Interrupt Level Model that reduced the amount of information given by the Hardware Level Model, the Interrupt Level State reduces the information given by the Hardware Level State to that needed by programmers.

The Interrupt Level State supplies information about individual processors and the communication channels between them. It does not deal with the parallel aspects of the machine which is domain of the Reconfiguration Level Model. The General Processor Model (Figure 7-3) as introduced in the Interrupt Level Model is used as a basis to analyze the state of processor components, and the Communication and Data Channel Model as shown in Figure 7-5 is used to examine processor communication.

The system programmer uses the information from the Interrupt Level State to decide which information has to be saved any time a program is disrupted. Such disruptions can be on a software level (e.g., kill and suspend system calls), or they can be caused by hardware (interrupts, bus errors). Here only hardware causes will be considered since the software disruptions depend on the operating system implementation and can thus not be part of the Interrupt Level State discussion.

The CPU State

As shown in Figure 7-3, the general model of any processor in the PASM system consists of a CPU, a Memory Mangement Unit, a set of I/O modules, and a set of memory modules. The CPU is the central component of any processor and will be discussed first. In the prototype, all CPUs belong to the Motorola MC68000 family. Even though the analysis is based on this particular microprocessor, the results are valid for a large class of CPUs since most properties exist in a similar way in any CPU. From a theoretical standpoint, the state of the CPU at any given time is of interest, and it is given by the value of all its internal registers. While some registers are accessible by programs (e.g., data registers), others are not (e.g., internal buffer registers). Therefore the complete state of the CPU cannot be saved, and it must be analyzed which variables have to be saved when a program is to be suspended and later resumed.

As mentioned above, the Interrupt Level State is most important at the time of a processing disruption, since state variables have to be saved at that time to facilitate program resumption. Processing disruptions can be classified into disruptions due to failures and legal disruptions. If a failure occurs (e.g., address error, illegal instruction, privilege violation), the processor executes an error handling routine after automatically saving some information on its stack. The information saved on the stack includes the program counter and the status register. These registers always change when the error routine is entered, and can thus be saved only by hardware. Since the disruption was due to a failure, the program will never resume. Thus the information saved on the stack can be discarded, and the user should be informed about the failure. No additional information from the CPU needs to be saved, but the operating system must abort the program gracefully. For example, it must close all communication channels used by the program. Thus it needs state information from other components as will be discussed later.

If a legal disruption occurs (e.g., interrupts and some kinds of bus errors), the CPU stacks some information as it does in the failure case. However, the value of the data and address registers is not saved automatically. Thus the state of the CPU during a legal disruption is given by the stack and the content of the address and data registers. When data and address registers are restored, and the stack content is loaded back into the CPU (by using a special instruction), the CPU state is identical to the state before the disruption.

Memory Management State

Many different memory management schemes have been devised [HwB84] and are thus considered for inclusion in various processors of PASM. In general, the state of the memory mangement can be given by a mapping from logical to physical addresses. In practice, a memory management unit (MMU) contains hardware registers that define this mapping, and the value of these registers is the MMU state. When a program is disrupted, the values of these registers must be saved if the program is to be resumed later, and new values have to be loaded.

The Memory State

The state of memory in general is given by the value of all memory cells. When a program is disrupted, the state of memory can be saved in two ways. The memory can be made inaccessible to other programs (e.g., through the use of the MMU), or it can be saved to secondary storage. When the program is to be resumed, the memory is made accessible to the program, or it is moved from secondary storage back to the memory. If a disruption due to a failure occurs, the memory content need not be saved and the memory space can be used by other programs.

The I/O Module State

Since I/O modules have widely varying characteristics, no general statements about their state can be made. Instead, the Parallel Port Board, which was also used in the discussion of Hardware and Interrupt Level Model, will be used as an example.

In the Interrupt Level Model of the Parallel Port Board a number of registers were shown. Some of these are initialized and never changed thereafter (e.g., data direction registers). Others, like the data transfer registers, change continuously. While the static registers can be considered part of the Interrupt Level Model, the value of the others determines the state of the board, and thus of the I/O module.

Not always is the state of the registers sufficient to describe the complete state of the interconnection. Consider as an example the network interface and the interconnection network. Once a path has been established through the network, data can be transferred along it. No register in the network interface, however, contains the routing information. Thus, if only the values of the registers on the interface are saved, the appropriate connection cannot be reestablished. Other information like routing information must be included in the variables saved at a program disruption so that the original situation can be restored.

The network itself needs special consideration since it is not an exclusive part of any individual PE but a shared resource. At any given time, the state of the network can be described by the current state of the finite state machines controlling the individual boxes (i.e, the setting of the interchange box), and by the state of the individual data and control lines. None of this information is accessible to programs, and can therefore not be saved. The routing tag of the message that established a path used by a program that is being disrupted is sufficient to reestablish the path when the program is resumed. Therefore the actual state of the network is irrelevant, and the routing tags used to control it can be used to describe the network state in a sufficient manner.

7.3.3 The Reconfiguration Level State

The Intelligent Operating System performs high level tasks like processor allocation, scheduling, and reconfiguration, and thus needs information much less detailed than the Interrupt Level State provides. The Reconfiguration Level Model as introduced above is a basis for the Reconfiguration Level State.

The Reconfiguration Level State is used in the Intelligent Operating System to orchestrate not only individual processors but the parallel aspects of PASM as well. In Figure 7-6, a Reconfiguration Level Model of the PASM system is shown. The Q MC Groups can work independent of each other or they can form larger partitions. The most fundamental state variable of the Reconfiguration Level State thus is the current machine partitioning. This variable specifies the partition each MC groups belongs to. Since the partitions can change dynamically, the variable might change frequently. With the knowledge of the current partitions and knowledge about the jobs running on the partition the operating system can decide which jobs to schedule, and where to allocate them. The Secondary Memory System load, unloads, and preloads data and programs the reside in the MC Groups. Its state is described by the tasks it is performing and still has to perform, and by the time each of these tasks will take. The System Control Unit coordinates the system activities, it set the partitioning hardware, and it informs the MC groups of the jobs they are to run. These variables can be considered as SCU state.

An MC Group Model is shown in Figure 7-7. The state of an MC Group is given by the partition it is assigned to, the current processing mode (SIMD or MIMD), by pending secondary memory access requests, and by the state of its shared memory (available shared memory). The state of the PEs of an MC group is simplified to the current use of a PEs memory. The available memory capacity is required to determine whether a new job fits into the memory. The available capacity of each memory buffer (Mem A, Mem B, Local Mem) is important so that the scheduler can utilize PASM's double-buffering scheme.

PASM can tolerate some kinds of faults. For example, the interconnection network is single-fault tolerant, and if a PE fails, the rest of the machine can continue to function. The location of faults, however, must be known to avoid them. In the interconnection network this involves rerouting of messages, and in the case of a PE fault the scheduler must avoid scheduling a job on the faulty PE. The faults existing in the system are thus another Reconfiguration Level State variable.

CHAPTER 8

SUMMARY AND FUTURE RESEARCH

In this thesis proposal, research already performed is described, and future research is outlined. One aspect of work focuses on the PASM Parallel Processing System hardware design. The design and implementation of the PASM prototype is described .n Chapter 4. The design and construction posed many challenges unique to parallel processing systems, and the dynamically reconfigurable aspects of PASM generated a need for much research and analysis into new techniques for architectural organization at the implementation level. The prototype will now become a testbed for performing research on many aspects of parallel processing. One aspect of this is software analysis. Many application algorithms for PASM have been developed, and the prototype facilitates running these programs. Parallel language and operating system development will also benefit from the availability of a testbed.

Due to its modular design, the prototype can be used to test new hardware concepts. One such concept is given in Chapter 5, where a new Network Interface Unit design that offloads communication overhead from the PASM PE CPUs to increase performance is proposed and analyzed. The NIU also provides a mechanism for fast shared memory accesses, a facility missing in the current prototype design. The NIU will be studied in more detail, and implemented and included in the prototype.

Even though the prototype follows the design concepts for a full size PASM, the scalability of its stru^cture is limited. An analysis in Chapter 6 shows tha. new avenues have to be explored to build a PASM system with 1,024 PEs. In that chapter, a packet-switched serial interconnection network is proposed that reduces wiring complexity, and a physical structure is shown that makes the interconnection of the PASM components feasible.

Chapter 7 focuses on theoretical aspects of PASM and reconfigurable systems. Research is proposed that demonstrates how to utilize PASM's reconfiguration capabilities to execute tasks in a dynamically changing environment. The characteristics of the PASM system are described by three levels of models, and the dynamic state is analyzed in three levels of state description. This provides the basis for an analysis of scheduling and reconfiguration. Changing requirements might necessitate the allocation/reallocation of resources to programs at run time. Methods to perform this reconfiguration will be studied. Ways to schedule tasks that have precedence relationships and whose run time is a function of the resources assigned to them will be examined. Special consideration will be given to the issue of data dependent dynamically changing precedences and priorities of tasks. This research will be directed toward the goal of an Intelligent Operating System for the optimal automatic reconfiguration of the PASM system.

LIST OF REFERENCES

[AdS82] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Transactions on Computers,* Vol. C-31, May 1982, pp. 443-454.

[AdS84] G. B. Adams III and H. J. Siegel, "Modifications to improve the fault tolerance of the extra stage cube interconnection network," *1984 International Conference on Parallel Processing,* August 1984, pp. 169-173.

[ArP76] R. G. Arnold and E. W. Page, "A hierarchical restructurable multimicroprocessor architecture," *Third Annual Symposium on Computer Architecture,* January 1976, pp. 40-45.

[BaB68] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Transactions on Computers,* Vol. C-17, August 1968, pp. 746-757.

[Bat76] K. E. Batcher, "The flip network in STARAN," *1976 International Conference on Parallel Processing,* August 1976, pp. 65-71.

[Bat77] K. E. Batcher, "STARAN series E," *1977 International Conference on Parallel Processing,* August 1977, pp. 140-143.

[Bat80] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers,* Vol. C-29, September 1980, pp. 836-844.

[Bat82] K. E. Batcher, "Bit serial parallel processing systems," *IEEE Transactions on Computers,* Vol. C-31, May 1982, pp. 377-384.

[BoD72] W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proceedings of the IEEE,* Vol. 60, April 1972, pp. 369-388.

[BrM85] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *1985 International Conference on Parallel Processing,* August 1985, pp. 782-789.

[Sei85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM,* January 1985, pp. 22-33.

[Com84] D. Comer, *Operating System Design, the XINU Approach,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[CrG72] B. A. Crane, M. J. Gilmartin, J. H. Huttenhoff, P. T. Rux, and R. R. Shively, "PEPE computer architecture," *IEEE Computer Society Compcon 72,* September 1972, pp. 57-60.

[CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing,* August 1985, pp. 531-540.

[Dav85] N. J. Davis IV, *Multistage interconnection networks: modeling, performance analysis, design, and fault location,* Ph.D. thesis, School of Electrical Engineering, Purdue University, 1985.

[DaH85] N. J. Davis IV, W. T.-Y. Hsu, and H. J. Siegel, "Fault location techniques for distributed control interconnection networks," *IEEE Transactions on Computers,* October 1985, pp. 902-910.

[DaS85a] N. J. Davis IV and H. J. Siegel, "The PASM prototype interconnection network," *1985 National Computer Conference,* July 1985, pp. 183-190.

[DaS85b] N. J. Davis IV and H. J. Siegel, "The performance analysis of partitioned circuit switched multistage interconnection networks," *Twelfth Annual Symposium on Computer Architecture,* June 1985, pp. 387-394.

[DeS85] E. J. Delp, H. J. Siegel, A. Whinston, and L. H. Jamieson, "An intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement,* November 1985, pp. 217-224.

[DuH73] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis,* John Wiley and Sons, New York, NY, 1973.

[Fly66] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[Fou81] T. J. Fountain, "CLIP4: progress report," in *Languages and Architectures for Image Processing,* M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 281-291.

[Fre61] H. Freeman, "Techniques for the digital computer analysis of chain-encoded arbitrary plane curves," *Proc. NEC,* Vol. 17, October 1961, pp. 421-432.

[GoL73] G. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *First Annual Symposium on Computer Architecture,* December 1973, pp. 21-28.

[GoG83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing an MIMD shared-memory parallel computer," *IEEE Transactions on Computers,* Vol. C-32, February 1983, pp. 175-189.

[HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, NY, 1984.

[JoC77] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans, "Software management of Cm* - a distributed multiprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 657-663.

[KaK79] S. I. Kartashev and S. P. Kartashev, "A multicomputer system with dynamic architecture," *IEEE Transactions on Computers,* Vol. C-28, October 1979, pp. 704-720.

[Kue86] J. T. Kuehn, *The PASM Parallel Processing System: Design, Simulation, and Image Processing Applications,* Ph.D. Dissertation, School of Electrical Engineering, Purdue University, 1986.

[KuS85] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems,* December 1985, pp. 603-612.

[KuS83] J. T. Kuehn, H. J. Siegel, and M. J. Grosz, "A distributed memory management system for PASM," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* October 1983, pp. 101-108.

[KuS82] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 International Conference on Parallel Processing,* August 1982, pp. 353-362.

[KuS85] J. T. Kuehn, H. J. Siegel, D. L. Tuomenoksa, and G. B. Adams III, "The use and design of PASM," in *Integrated Technology for Parallel Image Processing,* S. Levialdi, ed., Academic Press, San Diego, CA, 1985, pp. 133-152.

[Law75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers,* Vol. C-24, December 1975, pp. 1145-1155.

[MiR81] O. R. Mitchell, A. P. Reeves, and K-S. Fu, "Shape and texture measurements for automated cartography," *1981 IEEE Computer Society Conference on Pattern Recognition and Image Processing,* August 1981, pp. 367.

[Mot83] Motorola, *MC68451 Manual,* ADI-872-R1, Motorola, Inc., 1983.

[Nut77] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Fourth Annual Symposium on Computer Architecture,* March 1977, pp. 147-152.

[Pat81] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Transactions on Computers,* Vol. C-30, October 1981, pp. 771-780.

[Pea77] M. C. Pease III, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers,* Vol. C-26, May 1977, pp. 458-473.

[PfB85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," *1985 International Conference on Parallel Processing,* August 1985, pp. 764-771.

[SeU80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 National Computer Conference,* June 1980, pp. 631-641.

[Sie77] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Transactions on Computers,* Vol. C-26, February 1977, pp. 153-161.

[Sie79] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Transactions on Computers,* Vol. C-28, December 1979, pp. 907-917.

[Sie85] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.

[SiM81a] H. J. Siegel and R. J. McMillen, "Using the augmented data manipulator network in PASM," *Computer,* Vol. 14, February 1981, pp. 25-33.

[SiM81b] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer,* Vol. 14, December 1981, pp. 65-76.

[SiM78] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," *1978 International Conference on Parallel Processing,* August 1978, pp. 9-17.

[SiS86] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Tutorial on Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., to appear, 1986.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[Slo81] D. L. Slotnik, "Centrally-controlled parallel processors," *1981 International Conference on Parallel Processing,* August 1981, pp. 16-24.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[SwB77] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout, "The implementation of the Cm* multimicroprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 645-655.

[SwF77] R. J. Swan, S. Fuller, and D. P. Siewiorek, "Cm*: a modular multimicroprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 637-644.

[TuA83] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *1983 IEEE Computer Society Symposium on Computer Vision and Pattern Recognition,* June 1983, pp. 336-344.

[TuS81] D. L. Tuomenoksa and H. J. Siegel, "Application of two-dimensional bin packing algorithms for task scheduling in the PASM multimicrocomputer system," *Nineteenth Allerton Conference on Communication, Control, and Computing,* October 1981, pp. 542.

[TuS82] D. L. Tuomenoksa and H. J. Siegel, "Analysis of multiple-queue task scheduling algorithms for multiple-SIMD machines," *Third International Conference on Distributed Computing Systems,* October 1982, pp. 114-121.

[TuS84] D. L. Tuomenoksa and H. J. Siegel, "Task preloading schemes for reconfigurable parallel processing systems," *IEEE Transactions on Computers,* Vol. C-33, October 1984, pp. 895-905.

[TuS85] D. L. Tuomenoksa and H. J. Siegel, "Task scheduling on the PASM parallel processing system," *IEEE Transactions on Software Engineering,* Vol. SE-11, February 1985, pp. 145-157.

[WuF80] C-L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Transactions on Computers,* Vol. C-29, August 1980, pp. 694-702.

APPENDIX A.1

PARALLEL PROCESSING

This material will be published in

T. Schwederski, D. G. Meyer, and H. J. Siegel, "Parallel Processing," in *Computer Architecture: Concepts and Systems,* V. Milutinovic, ed., Elvesier Science Publishing Co., New York, NY, to appear, 1986.

Parallel Processing

Thomas Schwederski

David G. Meyer

Howard Jay Siegel

PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

1. Introduction

Many of today's scientific and industrial problems require enormous processing power. Examples of such computations are weather prediction, fluid dynamics studies, biomedical image processing, robot vision, plasma physics simulations, and VLSI design automation algorithms. To be able to solve these problems in a reasonable time, e.g., calculate a 24-hour weather forecast in less than 24 hours, very powerful computer systems are needed. Many efforts have been made to increase the processing speed of computers. For a long time, most speed increases were achieved by using faster components in the same basic computer structure. Vacuum tubes were replaced by discrete semiconductors, and these discrete components by integrated circuits. Core memory was replaced by integrated circuit memory chips. However, today's components seem to be approaching a practical speed limit. Recent developments of Gallium-Arsenide chips promise gate switching times in the picosecond range. At the present time, only superconducting elements offer faster speed, but superconducting computers are not expected to appear in the foreseeable future, if ever. A fundamental barrier that limits all computing speed is the velocity of light. Signals cannot propagate through a conductor faster than the speed of light, i.e., roughly one foot per nanosecond. Therefore modern high-speed CPUs tend to be very small (several cubic feet) in order to reduce the length of interconnections and by this, propagation delay.

Since faster components alone do not satisfy the current demands for high speed computing, other avenues are being explored. These methods utilize the fact that many computations in a program do not depend on each other, and can therefore be executed simultaneously. One such approach is overlap execution or pipelining. Pipelining is used extensively in modern supercomputers like the Cray-1 and Cyber 205. These machines achieve operating speeds of a few hundred million operations per second, but even these processing rates do not always satisfy current computational requirements. Another approach utilized to increase computational speed is the use of parallel processing, i.e., the use of a collection of computers coupled in some way to do faster processing. Many ways to combine computers have been proposed and implemented. They range from multiprocessor systems, where a set of identical or different computers are interconnected, to array computers, where a large number of processors execute the same instruction on different data.

In this chapter, parallel processing systems are introduced. Following a classification of parallel processing systems, software issues are discussed. Case studies describe some of the parallel computer systems that have been proposed or built.

2. Classification of Parallel Computer Systems

Many attempts to classify computers have been made; one of the best known is the classification by Flynn [Fly66], which is based on the number of concurrent instruction and data streams in a computer. An instruction stream is the sequence of instructions that are executed by a computer. The data stream is the sequence of data accessed to be processed by the instructions. Flynn distinguishes four classes:

SISD	Single Instruction stream - Single Data stream
SIMD	Singe Instruction stream - Multiple Data stream
MISD	Multiple Instruction stream - Single Data stream
MIMD	Multiple Instruction stream - Multiple Data stream

In the following subsections these four types will be discussed in detail.

2.1. SISD Machine Architecture

An SISD computer usually consists of a single processor connected to a single memory. It fetches its instructions consecutively, and fetches one data item at a time. Many standard computers fall into this class: microprocessors, minicomputers like Digital Equipment Corporation's VAX supermini computers, and mainframes like the IBM 360/370. Many modern SISD computers are pipelined to increase computing speed.

2.2. SIMD Machine Architecture

An SIMD computer typically consists of a control unit, N processors, N memory modules, and an interconnection network. The control unit broadcasts instructions to the processors and all processors that are currently active (enabled) execute the same instruction at the same time; this is the single instruction stream. Since each active processor executes the instructions on data stored in its associated memory module, a multiple data stream results. The interconnection network facilitates communication among the processors and memory modules. It is often referred to as an alignment or permutation network. There are many ways to implement such a network; for a detailed discussion of networks, refer to Chapter XX of this book.

SIMD machines are well suited for exploiting the parallelism of tasks such as matrix multiplication, which involve matrix or vector data, and problem domains such as image processing, where the same operation is performed on many different picture elements (pixels). SIMD machines are sometimes referred to as array processors. Examples of SIMD machines that have been built are the Illiac IV [BoD72], STARAN [Bat74], and MPP [Bat80, Bat82]. IBM's GF11 [BeD85] is an SIMD machine currently under construction. STARAN and MPP are overviewed in Section 5 of this chapter.

One way to view the physical structure of an SIMD machine is as a set of N processor/memory pairs called *processing elements (PEs)* interconnected by a network and fed instructions by the control unit (Figure CH.1). This configuration is called *processing-element-to-processing-element (PE-to-PE)* organization. The network is unidirectional and connects each PE with all or some subset of the other PEs. To move data between two PEs that are not directly connected, the data must be passed through intermediary PEs. For example, assume PE 0 is connected to PE 1, and PE 1 is connected to PE 2. To pass data from PE 0 to PE 2, PE 0 has to send its data to PE 1, and PE 1 passes them on to PE 2. The PE-to-PE configuration is used in the MPP system [Bat80, Bat82]. An alternative SIMD organization is the *processor-to-memory* organization. Here the network is placed between processors and memories as is shown in Figure CH.2. The network is bidirectional since it connects both memories to processors and processors to memories. One processor can transfer data to another processor through any memory to which both are connected. To move data between processors, the sending processor writes data into a shared memory and the receiving processor reads the data from that memory. As in the PE-to-PE configuration, multiple transfers involving multiple memories and processors may be necessary for transfers between processors which share no memory.

Variations of this scheme have been implemented. The STARAN processor, for example, uses a unidirectional network in a processor-to-memory configuration, as shown in subsection CH.5.4.

2.3. MISD Machine Architecture

Here a set of PEs works on the same data item but each PE executes a different instruction. One possible configuration is a macro-pipeline. Consider an MISD machine with N PEs, numbered 0 through N−1. In such a configuration the source data stream enters PE 0. PE 0 performs some operation on the data, e.g., it may multiply the data with a constant. The modified data is then passed on to the next processor which performs a different operation, e.g., a constant may be added. This process of passing data from PE i to PE i+1 continues until the result leaves PE N−1.

2.4. MIMD Machine Architecture

MIMD machines are commonly known as multiprocessors. An MIMD machine typically consists of N processors, N memory modules, and an interconnection network. Each of the processors follows its own instructions; these are the multiple instruction streams. Each processor also fetches its own data on which it operates; thus there are multiple data streams. In an MIMD machine, all processors operate asynchronously, which is more flexible than the lock-step execution of an SIMD machine. On the other hand, additional overhead is necessary if the processors of an MIMD machine need to be synchronized, since the synchronization needs to be done explicitly. In an SIMD system, synchronization is implicit by the lock-step operation. Due to this synchronization-cost/flexibility tradeoff, some computational tasks are better suited for the SIMD mode of parallelism, while others are better suited for the MIMD mode of parallelism.

As with an SIMD machine, two basic configurations can be distinguished: the PE-to-PE architecture (see Figure CH.3), and the processor-to-memory architecture (see Figure CH.4). Communication between processors is provided by the interconnection network. Since each processor is executing its own instructions, inputs to the network arrive independently (i.e., asynchronously), in contrast to an SIMD machine, where all inputs to the network arrive at the same time.

Examples of MIMD machines that have been built are Cm* [SwF77, SwB77, JoC77], the BBN Butterfly [CrG85], the Cosmic Cube [Sei85], and HEP [HaD85]. Proposed MIMD systems include the NYU Ultracomputer [GoG83] and the IBM RP3 [PfB85]. Cm* and Ultracomputer are overviewed in Section 5 of this chapter.

2.5. Multiple SIMD Machine (MSIMD) Architecture

An MSIMD machine is a parallel computer that can be dynamically reconfigured to operate as one or more independent virtual SIMD machines of various sizes. Similar to a standard SIMD machine, the processors, memories, and interconnection network can be configured in various ways. Figure CH.5 depicts a general model of an MSIMD machine with a PE-to-PE structure. The control units can be assigned to various PEs by setting the switch appropriately.

The MSIMD concept has various advantages over the standard SIMD machine. For example, faults can be detected by running the same program on more than one virtual machine and comparing the results, and fault tolerance is achieved by not using PEs that are known to be faulty. Multiple users can use the system at the same time; each user is assigned one virtual machine which runs programs independently of the other users. The machine is more efficient since a task that requires only a few PEs need not leave all others idle. Proposed MSIMD systems are MAP [Nut77a, Nut77b] and the original design for the Illiac IV [BaB68, BoD72].

≤.6. Partionable-SIMD/MIMD Machine Architecture

A *partitionable-SIMD/MIMD system* is a parallel computer that can be dynamically reconfigured to operate as one or more independent SIMD and/or MIMD machines. Such a system is very similar to an MSIMD machine; it differs in that its PEs cannot only follow an instruction stream broadcast by a control unit, but can also follow their own individual instruction stream. Thus the block diagram of the MSIMD machine also applies to the partitionable-SIMD/MIMD system. Like the MSIMD machines, the processors, memory, and interconnection network can be partitioned to form independent virtual machines. Each virtual machine can work in SIMD or in MIMD mode.

Examples of partionable-SIMD/MIMD systems are PASM [SiS81], TRAC [SeU80], DCA [KaK79], and PM4 [BrF79]. PASM and TRAC, both of which are in the prototype stage, are described in Section 5.

2.7. Chapter Overview

In this chapter, we will concentrate on SIMD and MIMD parallelism. Other approaches to parallel/distributed processing are discussed elsewhere in this book, and in books such as [Sto80, Bae80, HwB84, HoJ81, SnJ85, KaK82]. Software problems associated with SIMD and MIMD systems will be discussed in Section 3. Section 4 provides examples of parallel algorithms. Two MIMD systems (Cm* and Ultracomputer), two SIMD systems (STARAN and MPP), and two partitionable-SIMD/MIMD systems (PASM and TRAC) will be overviewed in Section 5.

3. Languages for Parallel Processing

3.1. Implicit Programming

Parallel computers can be programmed implicitly or explicitly. Implicit programming is done by using an intelligent, parallelizing compiler that accepts programs written in a conventional programming language like FORTRAN or Algol. The compiler then analyzes the program and determines which statements can be executed in parallel. For example, consider a program to add two vectors A and B:

```
FOR i=1 TO N DO C[i] := A[i] + B[i];
```

It is obvious that the addition statements are independent of each other, and therefore all additions can be executed in parallel on N processors. This example is typical of the operations performed by a parallelizing compiler. Since the order in which two statements I and J are executed in a sequential program is not preserved if I and J are executed in parallel, three conditions must be met if I and J are to be parallelized. Statement I must not write to locations that statement J reads, and vice versa, and statement I and J must not write to the same location if that location is used by a later statement.

In addition to parallelizing a program, the compiler must allocate the proper hardware to run the program. For example, in an MIMD computer the compiler must determine the optimum number of processors needed for the execution of the program.

It also has to allocate data in memory such that the parallel program executes efficiently and delays incurred from data transfers are minimized. This means that in a machine with PE-to-PE organization, all or most of the data a processor needs during program execution should be allocated in its own memory. In the processor-to-memory organization, data should be allocated such that processors get their data from different memory modules, sinre simultaneous accesses to a single shared memory unit by two or more processors can be served only sequentially and will therefore result in a higher access time.

3.2. Languages for MIMD Processing

If a parallel machine is programmed explicitly, the programmer must have available a programming language that allows the expression of parallelism in the program. If an MIMD computer is the target machine, a standard assembly or high level language like Pascal, C, or FORTRAN can be used for programming by adding operating system calls for interprocess communication and synchronization. The programmer has to decide which processor handles what particular subtask, and write the appropriate programs. The programs are then loaded into the processors needed for the task, and the operating system of the machine starts the task and coordinates activities. Many applications, especially in image processing, require a set of processors which all execute the same program. An example is an image processing algorithm in which each processor searches for features in a part of the image. In this case, each processor has its own set of data, but an identical program is executed by each processor. Only one program has to be coded, independent of the number of processors. Depending on the task, this may involve just MIMD operations, just SIMD operations, or both.

If the number of processors varies during task execution, the programming language has to be expanded to include special statements to allocate and deallocate processors. Conway [Con63] proposes the use of FORK and JOIN instructions. Whenever a FORK is encountered, a new process is created that proceeds independently from the old one. The old and the new processes can be run on different processors. If no processor is available, the new process must be queued until a processor becomes available. If multiprogramming capability is available, two or more processes can be run on the same processor in a time sliced fashion. Conway proposes three kinds of FORK: FORK A, J, K; FORK A, J; and FORK A. In this implementation, a counter is necessary to keep track of the number of concurrent processes. FORK A creates a new process; the old process continues to execute the instruction following the FORK, the new process starts at instruction A. The FCRK A, J, K instruction sets a counter in memory location J to the value K, then executes FORK A. The programmer is responsible for matching the number of FORKs in his program with the count specified in the FORK A, J, K, since the FORK A alone does not affect the counter. FORK A, J increments counter J by one and then executes a FORK A. This instruction is needed if the decision to fork is made at runtime.

The JOIN statement merges concurrent instruction streams. When a JOIN J is encountered, the counter at location J is decremented. The process that decrements the counter to zero executes the statements following the JOIN, all other processes are terminated and the processors which ran them are released and become available for other computations.

Figure CH.6 illustrates the use of the three different FORK and the JOIN statements. Process 1 executes a FORK A,100,3 which sets the counter in memory location 100 to three and starts a concurrent process (Process 2). Then Process 1 executes a FORK B,100, which increments the counter a location 100 by one and starts another process, Process 3. Meanwhile, Process 2 executes a FORK C, generating a fourth process, Process 4. Thus, four concurrent processes are now active. When a process encounters a JOIN 100, it decrements the counter. The process continues if the counter reaches zero, otherwise it ceases. FORK and JOIN are also useful for process synchronization; only after all forked processes have completed their program, does the main program continue – this is synchronization.

A more structured way to establish parallel subtasks than the FORK and JOIN instructions was proposed by Dijkstra [Dij68]: **parbegin** S1; S2; .... Sn; defines S1 through Sn to be statements (processes) which can be executed in parallel. The parallel execution is ended when a **parend** is found. Only after all processors executing the parallel statements have reached their **parend,** statements following the **parend** are executed. A program that needs to run two processes in parallel might have the following structure:

```
parbegin
process1 : begin
    statements for process 1
    end;
process2 : begin
    statements for process 2
    end;
parend
```

Another important synchronization mechanism is semaphores. A semaphore S is a shared variable which is accessible only through the operating system calls SIGNAL(S) and WAIT(S). A WAIT(S) reads the value of S and sets S to one in a single indivisible operation. If the value of S was found to be zero, the process executing the WAIT can continue. If the value was one, the process is suspended temporarily. A SIGNAL(S) resets S to zero, and reactivates all processes that have executed a WAIT(S) but were suspended. These processes all execute another WAIT(S); only the first process to do so will read a value of zero for S and therefore continue; all other processes read a one and are suspended again. Semaphores are useful in single CPU, multitasking systems where they are used for synchronization of processes which are running in a timesliced mode, and in multiprocessors, where each process runs on a different processor.

Semaphores can be implemented by the test-and-set instruction that is available in many computers. If a test-and-set is executed on memory location V, a 1 is returned if location V is already set; otherwise a 0 is returned and location V is set to 1. The operation is indivisible. Wait(X) is accomplished by a test-and-set on X. If X is 1, the process is suspended. If X is 0, the process sets the semaphore to 1 and execution can continue. Signal(X) is performed by simply setting the value of X to one. In most operating systems, this write operation will cause an interrupt so that the waiting process can resume execution.

Now consider how WAIT(S) and SIGNAL(S) can be used for processor synchronization. Assume processor A runs a program that needs at some time a result that a program running on processor B produces. Thus A needs to be synchronized with B when it needs B's result. This is accomplished by first allocating a semaphore S with a value of one. Immediately before A needs the result from B, A executes a WAIT(S), whereas B executes a SIGNAL(S) immediately after it has calculated the result required by A. If B had already computed the result when A executes its WAIT(S), the value of S was reset to zero by B's SIGNAL(S). Thus, B can immediately use the result and can continue. If, however, B's result was not yet available, A's WAIT(S) would find the value of S to be one. A would be suspended and reactivated only after B had executed the SIGNAL(S). In this way, synchronization is achieved.

These examples demonstrate two ways of how semaphores can be used. Further information is in [Hab76].

3.3. Languages for SIMD Processing

Parallelism of an SIMD computer cannot be expressed explicitly in any conventional programming language, but extensions of existing high level languages have been proposed, e.g., [Ste75, LaL75, Red73, Per79, MuS80, ReB80, Uhr81, ClS84, ClS85, KuS85b]. Such extensions can be machine dependent or machine independent.

An example of a machine dependent language is Glypnir [LaL75], which was specifically developed for the Illiac IV computer and is based on Algol 60. Illiac IV was an SIMD machine with 64 PEs, designed in the 1960's [BaB68, BoD72]. Its fixed interconnection network connects PE i to PEs i+1, i−1, i+8, i−8, all modulo 64, providing a mesh-type interconnection. In Glypnir, variables for PEs and for the control unit can be defined. A PE variable has an element in every PE and is called a super word *(sword)*, which always has 64 elements. For example, the statement

```
PE REAL A, B
```

declares the variables A and B to be swords.

```
PE REAL C(20)
```

declares C to be an array of 20 elements in each of the PEs. By using boolean expressions, PEs can be selectively enabled or disabled. Consider, for example, the statement

```
if <boolean expression> then
    <statement1>
else
    <statement2>.
```

The boolean expression will be evaluated in all processors, and the processors finding a true value execute statement1. The others do not participate in the instruction stream broadcast for statement1 but execute only statement2.

Actus [Per79], a language based on Pascal, is an example of a machine independent language. It was designed for SIMD machines of arbitrary size. A variable can be declared as parallel, and the extent of parallelism must be specified for each parallel variable. In the range specification of a parallel variable declaration, a colon denotes that a dimension is to be accessed in parallel. For example,

var *array1:* **array** [*1:m,1..n*] **of** *integer*

declares an array of m by n elements; m elements can be accessed at a time if they are placed in different processors. The underlying operating system takes care of reconfiguring the computer into a machine that can handle the specified parallel variable. **if, case, while,** and **for** statements are expanded for use with parallel variables. Alignment operators are also provided. For example, a **shift** operator moves data within the declared range of parallelism, a **rotate** shifts data circularly with respect to the extent of parallelism. Shift and rotate are implemented by using the processor interconnection network.

A programming language to handle both the MIMD and SIMD modes of operation is proposed in [KuS85b]. It is based on the C programming language. The SIMD features include declarations of parallel variables and functions; an indexing scheme to access and manipulate parallel variables; expressions involving parallel variables; extended control structures using parallel variables as control variables; and functions for PE allocation, data alignment, and I/O. The language supports simple parallel data types like scalars and arrays as well as structured data. For example,

```
parallel [N] int a;
parallel [N] char line[MAXLINE];
struct node {
        char *word; struct node *next;
} parallel [N] nodespace[100], *head;
```

declares, for each of N virtual processors, an integer "a," an array of MAXLINE characters, an array of 100 nodes, and a node pointer. Indexing along the parallel dimension can be done by using selectors; they enable or disable subsets of the N virtual processors. Functions to send data between processors are provided. They are not part of the language, but library routines, thus avoiding machine dependence. To facilitate MIMD mode, a few new features and keywords are added, supporting **parbegin, parend,** shared data, synchronization, etc. A preprocessor recognizes these constructs and translates the parallel algorithm into a standard serial C program, which can then be compiled by a standard C compiler and loaded into the program memories of the processors of the MIMD machine.

One of the ways this language implements selectors which is based on the use of a *PE address masking* scheme [Sie77]. This scheme uses an $n=\log_2 N$-position mask to specify which of the N PEs will be activated. Each position of the mask will either contain a 0, 1, or X ("don't care") and the only PEs that will be active are those whose addresses match the mask: 0 matches 0, 1 matches 1, and either 0 or 1 matches X. Square brackets denote a mask, superscripts are used as repetition factors. For example, "$[X^{n-1}0]$" activates all even numbered PEs and "$[0^{n-i}X^{i}]$" activates PEs 0 to $2^{i}-1$. A ***negative PE address mask*** is similar to a regular PE address mask, except that it activates all those PEs which do not match the mask. Negative PE address masks are prefixed with a minus sign to distinguish them from regular PE address masks. For example, for N=8, "[−0X1]" activates all PEs except 1 and 3.

4. Examples of Parallel Algorithms

As mentioned in the previous section, programming of parallel computer systems can be done explicitly or implicitly. Implicit programming will not be discussed here, since only the compiler takes the parallelizing action. Instead, explicit programming examples will be emphasized to illustrate the techniques of restructuring a problem to make :t suitable for parallel execution. In this Section, SIMD algorithms, associative processing algorithms, and SIMD/MIMD algorithms will be presented.

4.1. SIMD Algorithms

As an example of a simplè algorithm, consider the ***smoothing of an image*** [SiS81]. The algorithm described here smooths a gray level input image. The algorithm has "I" as an input image and "S" as an output image. Assume both I and S contain 512-by-512 *pixels* (picture elements) for a total of 512^2 pixels each. Each point of I is an eight-bit unsigned integer representing one of 256 gray levels. The gray level of each pixel indicates how "dark" that pixel is, where 0 means white and 255 means black. Each point in the smoothed image, S(i,j), is the average of the gray levels of I(i,j) and its eight nearest neighbors, I(i−1,j−1), I(i−1,j), I(i−1,j+1), I(i,j−1), I(i,j+1), I(i+1,j−1), I(i+1,j), I(i+1,j+1). The top, bottom, left, and right edge pixels of S are set to zero since their corresponding pixels in I do not have eight adjacent neighbors.

Consider how this could be implemented on an SIMD machine with N = 1024 PEs, logically arranged as an array of 32-by-32 PEs as shown in Figure CH.7. Each PE stores a 16-by-16 subimage block of the 512-by-512 image I. Specifically, PE 0 stores the pixels in columns 0 to 15 of rows 0 to 15, PE 1 stores the pixels in columns 16 to 31 of rows 0 to 15, and so on. Each PE smooths its own subimage, with all PEs doing this simultaneously. At the edges of each 16-by-16 subimage, data must be transmitted between PEs in order to calculate the smoothed value. The necessary data transfers are shown for PE J in Figure CH.7. Transfers between different PEs can occur simultaneously; e.g., when PE J–1 sends its upper right corner pixel to PE J, PE J can send its upper right corner pixel to PE J+1, PE J+1 can send its upper right corner pixel to PE J+2, etc.

In order to perform a smoothing operation on a 512-by-512 image by the parallel smoothing of 1024 subimage blocks of size 16-by-16, $16^2 = 256$ parallel smoothing operations are performed. As described above, the neighbors of the subimage edge pixels must be transferred in from adjacent PEs. The total number of parallel data element transfers needed is $(4\times16) + 4 = 68$: 16 for each of the top, bottom, left side, and right side edges, and four for the corners (see Figure CH.7). The corresponding serial algorithm needs no data transfers between PEs, but $512^2 = 262{,}144$ smoothing calculations must be performed. If no data transfers were needed, the parallel algorithm would be faster than the serial algorithm by a factor of $262{,}144/256 = 1024 = N$. If the inter-PE data transfer time is included and it is assumed that each parallel data transfer requires at most as much time as one smoothing operation, then the time factor improvement is $262{,}144/324 = 809$. The inter-PE transfer time approximation is a conservative one. Thus, the overhead of the 68 inter-PE transfers that must be performed in the SIMD machine is negligible compared to the reduction in smoothing operations from 262,144 to 256.

As mentioned above, the edge pixels of S are not smoothed. This creates an additional (although negligible) overhead which is to disable the appropriate PEs when the zero values are stored for these edge pixels.

The method of ***recursive doubling*** is a technique used to speed up parallel computations in a manner that is not straightforward. Consider the problem of summing up all elements of a vector. In a serial computer, a variable would be initialized to the value of the first element of the vector, and all subsequent elements would be added to this variable. If the vector contained N elements, N−1 additions are required. Assume that on a parallel computer the vector is distributed among processors, i.e., each of N processors holds one element. If the serial method would be used on the parallel machine, one processor would accumulate the result, and all other processors would send their element values to this processor. Clearly, this still requires N−1 additions, all performed by the accumulating processor. Thus, no speedup is achieved by using a parallel machine.

The recursive doubling procedure is illustrated in Figure CH.8. Assume that N is a power of 2; in the Figure, N=8. In the first step, all odd numbered PEs (i.e., 1, 3, 5, 7) send their vector element to an even numbered PE (i.e., 1 to 0, 3 to 2, etc.). All even numbered PEs then add the value they received to their own vector element, forming N/2 (=4) partial results. The odd numbered PEs do not participate and are disabled. In the next step, this procedure is repeated. PE 2 forwards its partial result to PE 0, while PE 6 sends its result to PE 4. PEs 0 and 4 add the new value to their respective partial results, and all other PEs are disabled. In the last step, PE 4 sends its partial result to PE 0. PE 0 adds this value to its own partial result, and the sum is found.

The overall procedure required three transfers and three additions. The example can be expanded to higher values of N. For N a power of two, $\log_2 N$ additions and data transfer steps are required. This is a significant improvement over the serial algorithm.

Examples of more complex SIMD algorithms for image and speech processing include [JaM85, KuF85, SiS82, WaS82, YoJ85, YoS82].

4.2. Associative Processing

In this subsection, associate processing, a special case of SIMD processing, will be introduced. An associative processor achieves its capabilities through the use of a *Content Addressable Memory (CAM).* Such a memory is not addressed like a standard random access memory (RAM). Instead, the "address" to the CAM is a value, and all CAM cells that contain this value in a particular field will set a flag indicating that their content matches the input value. A cell may be, for example, 1K bits long. Obviously only part of the cell should be matched since otherwise no information apart from the already known input value could be provided by the matching cell. This basic CAM operation is illustrated in Figure CH.9. Assume the match register is initially reset to all zeroes. The CAM memory array contains some information about employees of a company, i.e., the employee name, his or her salary, and the time employed. Each cell could of course contain more information fields than those shown here. If all employees whose salary is higher than $2000 are to be found, a comparand is presented to the memory cells which contains "$2000" in the salary field and "don't care" elements (represented as "X") in all other fields. A controller (not illustrated here) must specify the operation "greater than" to the memory cells; all cells will then compare the salary field of the comparand with their own salary value simultaneously. If a salary higher than $2000 is found, a match bit in the match register will be set to one; thus all persons with a salary higher than $2000 will have been found in a single CAM cycle. If all salaries between $2000 and $3000 are to be found, two steps are necessary. The first step is the one described above, and all persons with their salary higher than $2000 are found. To find all persons in the desired range, only those cells that have already found a match should participate in the search for salaries less than $3000. It is therefore desirable that the match register can also be used to select a subset of cells to participate in the operation, making searches like the above possible.

The hardware structure is shown in Figure CH.10. All information is encoded in binary. Thus all fields of the memory array can now be represented as a homogeneous block of memory bits. The rows of this array hold the information records, e.g., the information about the employee as in the above example. Thus each element of a column of the array has the same meaning, e.g., the least significant bit of the employee's salary. The value with which the content of the memory cells is to be compared is written into one or more fields of the comparand register. The match register holds a zero if the value of the bit column is to be ignored and a one if the bit column is to participate in the operation, i.e., the mask register selects the fields to be used for the search. If a match is found during the operation, the match register bit is set to one.

Some operations, such as printing of all matching names, necessitate a sequential reading of values. If there is more than one match, the information must be retrieved row by row. A match resolver accomplishes this task by selecting one of the matching rows for retrieval, usually the "first," i.e., topmost, match. The corresponding match register bit is reset to zero after the retrieval, and then the resolver will select the second element, etc.

Writing to an associative memory is done in a fashion similar to reading. First some cell is selected for writing by matching it with some comparand, the match resolver selects one specific cell, and the appropriate word is then written into the selected cell. The mask register may be used during the write operation to modify only part of the cell, e.g., just change the salary of an employee. It is also possible to write to all cells which match the comparand simultaneously.

Searches, such as described above, and arithmetic operations are usually implemented as a series of more primitive hardware operations. A simple example is the addition of one to a 16-bit integer in each CAM cell. Let B denote a 16-bit integer field in all CAM cells, i.e., each cell has its own B field value. Let B_j denote bit j of integer B. C denotes a one-bit carry bit field in each CAM cell. Figure CH.11 illustrates the algorithm to add one to each B. The carry bit C is initially set to one. Then each bit position of the integer is examined in the main loop. If the carry in combination with the integer bit require a change, this change is performed by a write operation until all bit positions of the integer have been examined. The comparisons of the previous example, e.g., the test for greater $2000, are also performed by a series of bit equality matches.

This algorithm shows the performance advantage CAMs algorithms have over sequential computers. If there are N integers stored in N CAM cells, each integer with b bits, the time to add one to each of the N integers is proportional to b and independent of N. In a sequential computer, the time to execute the algorithm will be proportional to N and independent of b. Thus, in general, if N is larger than b, the CAM will operate faster.

Two basic associative memory structures can be distinguished. A *bit-parallel* CAM performs the search operations in parallel; i.e., if 20 bits are to be matched, this match will take a single CAM cycle. This requires that each bit cell of the memory array has the appropriate circuitry necessary for the search operations associated with it. Therefore large bit-parallel memories are very complex and expensive, and are thus not widely used. With today's advanced integrated circuit technology, large integrated bit-parallel CAMs may soon be seen. The bit-parallel organization contrasts with the *bit-serial* CAM as follows. Here a match is performed one bit at a time: if 20 bits are to be matched, 20 memory cycles are needed. During each of the cycles, one bit slice is compared with the appropriate bit in the comparand register. Bit serial CAMs are slower than their bit-parallel counterparts, but they are much cheaper; in fact, they can be built from standard random access memory, as shown in the discussion of the STARAN computer in Section 5. For a more detailed description of CAMs and their capabilities, see [Fos76].

4.3. SIMD/MIMD Algorithms

Consider an SIMD/MIMD parallel implementation of *contour extraction* [KuS85a, TuA83]. A contour in an image is the boundary of some feature of the image. Two algorithms from a contour extraction task are *edge-guided thresholding (EGT)* and *contour tracing*. The EGT algorithm is used to determine a set of optimal thresholds for quantizing the image, i.e., translating each pixel from a gray level value to either black (1) or white (0) [MiR81]. The contour tracing algorithm uses the set of thresholds to segment the image and trace the contours. It is assumed that the image to be processed is distributed among the PEs as in the smoothing example.

The EGT algorithm consists of three major steps. First, the Sobel edge operator [DuH73] is used to generate a "Sobel-image" in which gray levels indicate the magnitude of the gradient. The magnitude of the gradient represents the amount of change in the gray level of the original image from one pixel to the next. A figure of merit that indicates how well a given threshold gray level matches edges in the edge image is then computed for every possible threshold. Finally, the maximum value of the figure of merit function is chosen to determine the threshold level. This is done for each PE's subimage independently. Thus, the threshold levels may differ from one subimage to the next. The window-based Sobel operator calculations and inter-PE communications used in the SIMD EGT algorithm are very similar to those discussed earlier for the image smoothing algorithm.

The MIMD contour tracing algorithm has two phases. In Phase I, PEs segment their subimages based on the threshold value each calculated using the EGT algorithm, i.e., each PE converts all pixels of its subimage from gray level values to either black or white. Within each subimage, all contours (collections of connected black pixels) are traced Some will be closed (complete), others will extend into neighboring subimages (partial). In Phase II, partial contours traced during Phase I are connected.

In Phase I there is no PE-to-PE communication. Each PE creates a segmented subimage for a particular threshold T by assigning a value of one to subimage pixels having a gray level greater than or equal to T, and a value of zero to the others. Contour tracing starts with each PE scanning the rows of its segmented subimage beginning with the first pixel of the top row. Scanning stops when a start point of a new contour is found. A start point is a pixel with a value one which has a zero-valued neighbor to either or both sides. Contours are traced in either a clockwise or counterclockwise direction and the Freeman direction codes [Fre61] of the "chain" of pixels are recorded. When a pixel from an adjacent subimage would be required to determine the next direction of the contour, a point of indecision is reached. Such a point is recorded as an end point, and the algorithm returns to the start point to trace the contour in the opposite direction until another point of indecision is reached. Closed contours that are contained within a subimage are traced completely during Phase I.

In Phase II, each PE attempts to connect its partial contours to those located in neighboring PEs. PEs consider each partial contour's end point in turn and try to find a possible extending contour in a neighboring PE. Once such an extended contour is found, the process is repeated, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended. A protocol employing semaphores is necessary to prevent more than one PE from trying to use the same partial contour as an extending contour at the same time. Phase II is complete when all contours have been connected.

The contour extraction demonstrates the advantages of an SIMD/MIMD machine as contrasted to a pure SIMD machine or pure MIMD machine. The EGT algorithm can be more efficiently executed in SIMD mode, while the tracing algorithm can be more efficiently executed in MIMD mode. Thus, the ability of the PE to operate in either SIMD or MIMD mode allows the most appropriate type of parallelism to be employed by each algorithm in the task.

5. Case Studies of Parallel Processing Systems

5.1. Introduction

In this section, examples of SIMD and MIMD machines are presented. Brief case studies overviewing six parallel processing systems that have been built or have prototypes under development are given. Different approaches to processor design, processor-to-processor and processor-to-memory communication, and system organization are demonstrated.

5.2. The Cm* Multiprocessor

The Cm* computer was built at Carnegie-Mellon University during the middle 1970's [SwF77, SwB77, JoC77]. Cm* is a shared memory multicomputer consisting of a set of computer modules (Cm); each Cm has its own memory and can access the memories of all other Cms through a two-level shared bus system. To provide a distinct address for all memory cells in the system, all memories are pa·t of one large address space of 2^{28} bytes.

The basic Cm* block diagram is shown in Figure CH.12. Each Cm in this diagram consists of a commercial LSI-11 microcomputer CPU, a private memory, I/O devices and an *Slocal.* Groups of up to fourteen Cms are interconnected through a *Map Bus.* The Slocal determines whether a bus request issued by the CPU is destined for the local memory or for memory in a different Cm (which must be sent to the Map Bus) and routes the request accordingly. It also accepts memory requests from the Map Bus and passes the result of the reference back to the Map Bus. The Map Bus is controlled by the *Kmap* that also routes memory accesses to Cms outside the local group if needed. The group of Cms, the map bus, and the Kmap make up a *cluster.* Clusters are connected by intercluster busses.

A detailed picture of the Kmap is shown in Figure CH.13. The Kmap consists of a *Kbus* that arbitrates and controls the Map Bus, the *Pmap* that maps the CPU addresses into the 2^{28}-word total address space and vice versa, and the *Linc,* the interface to the intercluster busses.

The Kmap is responsible for handling intracluster and intercluster references. In an intracluster reference, a Cm accesses a memory in the cluster the Cm belongs to, and in an intercluster access, the Cm reference is routed to another cluster. Consider an intracluster read operation. The source Cm issues a memory request, which is received by the Kbus. The Kbus places the request in the run queue, where it is read by the Pmap. The Pmap performs an address translation and places the translated address in the out queue. The Kbus reads the translated address and places it onto the Map Bus, thus forwarding it to the the destination Cm. There the Slocal performs the required memory access. The destination Cm then issues a return request to the Kbus, and the Kbus completes the intracluster reference by strobing the data into the source Cm. A write is different in that the data is forwarded with the address, and an acknowledge is passed back instead of the data.

An intracluster reference is more complex. Consider an intercluster read. When the source Cm has issued an intercluster request, the Kbus latches the destination address and places it into the run queue. The Pmap reads the queue, translates the address, and places the new address into one of the Port Send Queues depending on which intercluster bus is used. The Linc reads the Port Send Queue and places the request onto the appropriate intercluster bus. Since an intercluster memory reference takes up to 20 microseconds, it is not efficient start a transaction and then wait for the return to arrive. Instead, the Kmap suspends the current access until a return message is received. The information necessary to reactivate the access is stored in the Pmap.

At the destination Kmap, the intercluster request is received by the Linc and is placed into the Service Queue. This queue is read by the Kbus, which translates the address via Run Queue, Pmap, and Out Queue. The appropriate memory access is then performed at the destination Cm. The return message (which includes the data) is routed back to the source cluster via Kbus, Run Queue, Pmap, Port Send Queue, and Linc. At the source cluster, the Linc receives the return message and places it into the Return Queue. The Kbus reads the message from this queue, and places it into the Run Queue. The Pmap then reactivates the access, places the necessary information into the Out Queue, and the Kbus strobes the data into the appropriate memory location, thus completing the intercluster reference. As with intracluster references, a write is different in that the data is forwarded with the address, and an acknowledge is passed back instead of the data.

As seen from the above discussion, a memory reference can either be local, intracluster, or intercluster. Due to the rather slow access time to non-local memories, performance of the system degrades if not most of the references are local, especially if many Cms are grouped in a cluster. For example, if all memory references are local, performance increases approximately linearly with the number of processors in a cluster, as could be expected. If no intercluster references are executed, and 40% of the references are inter-Cm references, the performance of a cluster with 16 Cms is only approximately twice as high as a cluster with four Cms. This is a rather dramatic decrease in performance, and clearly shows that shared bus systems have only limited application domains. In this limited domain of programs with mostly local references, a system like Cm* has the advantage of easy expandability (Cms can be added to clusters, and new clusters can be added to the system) and low cost (most components like CPU, memory and I/O are commercial products).

5.3. The NYU Ultracomputer

The NYU Ultracomputer is a proposed design of a large scale shared memory MIMD machine [GoG83, EdG85]. It could have as many as 4096 processors connected to 4096 memories through a multistage cube interconnection network; thus the basic Ultracomputer configuration is of the processor-to-memory type. One of the most important features of the Ultracomputer is the fetch-and-add instruction, denoted *F&A(V,e),* which provides a simple and efficient way for processor synchronization and is supported by hardware. V is an integer variable and e is an integer expression. The execution of an F&A instruction returns the old value of V and adds to V the value of e. The instruction is indivisible, i.e., while the F&A operation is being performed on V by one processor, no other processor can access V. If two processors A and B try to execute a F&A operation on the same variable at the same time, the effect will be as if one processor preceded the other in time. For example, processor A can receive the old value of V and V will be incremented by the value of e of processor A. Processor B will then receive the new value of V, and V is incremented by the value of e of processor B.

The F&A operation can be used to solve a wide variety of synchronization problems; the following example illustrate‿ its use. Suppose N processors are to be used to each perform a complex operation on an N element array, e.g., calculate the tangent. The first task is to associate one processor with each of the N array elements. This is easily accomplished by initializing a shared variable V1 to zero, and then having each of the N processors execute a F&A(V1, 1). Each processor will receive a distinct value for V1 in the range from 0 to N−1; thus each processor knows the array element it is supposed to handle. Only after all processors have completed their task, processing that utilizes the result of the computations can resume. To ensure this, a second shared variable is set to N−1 before the parallel execution starts. After a processor has completed its task, it executes a F&A(V2,−1). If the returned value is not 0, other processors are still working on their assigned array element. Therefore the processor simply terminates working on this problem and can be reassigned to other processes. The processor that receives a value of *zero* is the last one to complete the task, and this processor continues execution of the program. The example shows similarities between the F&A operation and the FORK and JOIN commands described earlier in this chapter.

The basic fetch-and-add operation can be extended to a fetch-and-ϕ operation, where ϕ is some function. Here the old value of V is fetched and replaced by the value of $\phi(V,e)$. An example is the fetch-and-or operation. The value of a variable is fetched, and the value is "or"ed with e. If V and e are binary quantities, the operation is identical to the previously described test-and-set operation.

If ϕ is the function f_1, such that $f_1(V,e) = V$, a regular load operation is executed. The value of e is not needed and therefore need not be forwarded to the fetch-and-ϕ hardware. A normal store can be accomplished by using the function $f_2(V,e) = e$ and ignoring the value passed back. Thus, a computer providing only fetch-and-ϕ operations can execute all standard memory accesses.

The basic structure of the Ultracomputer is illustrated in the block diagram in Figure CH.14. The N processors are not necessarily identical as in many other designs; at least some of them could be special purpose machines for FFT computations or matrix multiplications, or they could be I/O processors interfacing the system with the outside world. Due to the large number of processors (up to 4096), most of them would very likely be identical one-chip processors with some local memory (cache). The processors interface to the interconnection network through a processor network interface *(PNI)*, and the network is connected to each of the N memories through a memory network interface *(MNI)*.

Much of the processing power of the Ultracomputer originates from the sophisticated interconnection network. The network design is a message-switched multistage omega network that also supports F&A operations and can be expanded to support fetch-and-ϕ. The omega network is a member of the multistage cube family of networks that is described in Chapter XX. The network has $\log_2 N$ interconnected stages, where each stage consists of N/2 2-input, 2-output switches. By connecting a switch's inputs and output paths through the network can be established. In this chapter, only the implications of the message switching and the F&A hardware will be discussed.

A message-switched network contrasts to a circuit-switched network in the following manner. *Circuit switching* implies the establishing of a complete path between the source (i.e., the processor) and the destination (i.e., the memory). Once such a path is established, subsequent transfers along the path can be accomplished at high speed and efficiently. Problems arise, however, from the possibility of blocking (a path might not be possible since part of the path is used by another, already established path). A *packet-switched* network, often referred to as *message-switched,* does not establish a complete path from source to destination. Instead, data and routing information are collected into a packet, and this packet makes its way from stage to stage, releasing links and interchange boxes immediately after using them. Thus, a packet uses only one interchange box at a time. Since the packets are passed from stage to stage, the travel time of a packet is longer than a transfer along the direct connection in a circuit-switched system. If the processor does not wait for a data item to return after sending a request to the network, but resumes processing – perhaps on a different task – this speed disadvantage need not be serious.

Hardware to implement the F&A operation is provided at two places: adders in the MNI and intelligent switches in the network. When a F&A(Y,e) has passed through the network and reaches the MNI, the value of Y is fetched from memory. This value is passed back through the network to the source of the request, and the value of Y is added to e. The sum is then written back to memory, resulting in the desired F&A operation.

An adder and a local memory is associated with each switch in the network. Whenever two requests for the same memory location, e.g., F&A(X,e1) and F&A(X,e2), arrive at a switch, the switch stores e1 in its memory, adds e1 and e2, and passes a request for F&A(X,e1+e2) on to the memory. The memory updates the variable X by adding e1+e2 to it, and passes the original value of X back to the network. When this return message reaches the above switch, the switch passes the original value of X back to the location that requested F&A(X,e1), adds the stored value of e1 to the value of X, and passes the result back to the second source.

If requests to the same memory location that were combined as described above meet at another switch, they can be combined further, thus combining combined requests. In this manner, multiple requests to the same memory location can be served in the time of a single request. Interprocessor coordination is therefore not serialized. With appropriate additional hardware the network can be generalized to facilitate associative fetch-and-ϕ operations.

The Ultracomputer promises to be an extremely powerful computer. It was estimated that a 4096 PE machine could be packaged into a 5×5×10 ft enclosure using 1990 technology. A 64 processor prototype is currently under development.

5.4. The Massively Parallel Processor (MPP)

The MPP, the Massively Parallel Processor [Bat80, Bat82] is a very powerful SIMD computer that was designed by Goodyear Aerospace Corporation and delivered to NASA in 1982. Its processing power originates from 16,384 bit-serial processing elements; the machine can perform up to 6,553 million eight-bit integer additions per second.

The basic structure of MPP is shown in Figure CH.15. The Array Unit (ARU) performs array computations under the control of the Array Control Unit which also executes scalar arithmetic. The overall flow of data and control in the MPP is handled by the Program and Data Management Unit, which is also used to develop programs and perform diagnostic functions. A staging memory and two 128-bit wide I/O interfaces transfer data to and from the ARU through two switches. The staging memory interfaces the ARU with magnetic tape, disk, terminal, line printer, and the external host computer.

The ARU consists of an array of 128-by-132 PEs, of which a 128-by-128 square array of PEs is enabled at any given time. The PEs are simple bit-serial processors with 1024 bits of memory each. A subarray of 4-by-2 processors (without the memory) are packaged into a VLSI chip. Four of the columns of the array are included for fault tolerance; four columns were chosen due to the 4-by-2 organization of PEs in the VLSI chips. If no fault exists in the array, an arbitrary set of four columns is disabled.

When a fault occurs, the set of the four columns the faulty PE belongs to is disabled, and processing can continue at full speed. Since a fault usually corrupts data, the currently executed program is restarted from the beginning or rerun from a checkpoint at which all data and the machine status were known.

The interprocessor communication in MPP is of the PE-to-PE type, and no shared memory is provided. Due to the very large number of processing elements, the simplicity of each PE, and the expected computational tasks, a rather simple interprocessor communication network was chosen: each processor is connected to its four nearest neighbors. The top and bottom connections can either be left open or be connected within each column. The left and right edges can be left open or be connected within each row, in an open spiral, or in a closed spiral. For the open spiral, PE i is bidirectionally connected to PE $i+1$, $0 \leq i \leq 16{,}382$ (i.e., PE 0 and 16,383 are not connected). In the closed spiral mode, PEs 0 and 16,383 are also connected. These modes of the edge connections can be chosen under software control. The choice of edge connections will depend on the algorithm being executed. In some cases, multiple data transfers may be required to move data to their intended destination.

The MPP PEs are bit-serial processors; a diagram of a PE is shown in Figure CH.16. The speed of the PEs is one operation per 100 ns. Six one-bit registers, the A, B, C, G, P, and S register, are available. They communicate with each other and with the random access memory through a data bus (D). Arithmetic operations are performed by a one-bit full adder and a shift register of programmable length. The interconnection network interfaces to the P register, which has four bidirectional paths to each of its nearest neighbor PEs. To route information, for example, from left to right, each PE shifts the contents of its P register into the P register of the PE to its right, and thus receives the information from its left neighbor. Additional logic associated with the P register (see Figure CH.16) can perform all boolean operations on the content of the P register and the bit currently on the data bus. The result is left in the P register. The adder sums the content of the A and P registers and the carry in the C register; it leaves the least significant bit of the result in the B register, and the carry in the C register. The length of the shift register can be programmed to be 2, 6, 10, 14, 18, 22, 26 and 30 bits. The A and B registers can be thought of as additional elements of the shift register; the total shift register length is thus a multiple of four. These simple components are sufficient to perform all basic arithmetic operations: addition, subtraction, multiplication, division, and floating point operations.

In many applications (such as in the smoothing example), PEs have to be disabled. To disable PEs, the G register is provided. If a *masked* instruction is executed, only those PEs with a 1 in their G register participate in the execution.

A comparator speeds up certain algorithms, e.g., normalization of floating point numbers. Global minimum value and maximum value searches and other global operations can be performed by the Sum-Or-Tree, which is a tree of inclusive-or gates. It has inputs from all PEs, and its output is connected to the Array Control Unit.

The important role of I/O is handled by the S-register of each PE. Together, the 128^2 S-registers form a plane that can shift data. Each S-register operates independently from the rest of the PE. An S-register sends its 1-bit content to its right neighbor and receives the 1-bit content from the S-register that is its left neighbor, i.e., 128-bit columns of data are shifted across the S-register plane. The 128 bits of data that are simultaneously shifted into the S-registers of the leftmost column of PEs originate from the previously mentioned input switches (Figure CH.16). This shifting is done without interrupting normal processing. After 128 shifts cycles, a complete 128-by-128 bit plane is loaded into the S-registers. Processing is interrupted for a single 100ns cycle and this complete data plane of 128-by-128 bits is transferred from the S-registers into the PE memories, each S register sending one bit to its PE. If necessary, all PEs can simultaneously move a memory bit into their associated S-register in a second cycle. Then a new plane of 128-by-128 bits can be shifted in and the old plane can be shifted out at the same time. The data that is shifted out of the rightmost PEs is accepted by the output switches, one column (128 bits) per cycle (Figure CH.16). This method of overlapping shift in and shift out provides a very efficient way to get data into and out of the MPP PEs.

Compared to state-of-the-art microprocessors, like the Motorola MC68020 32-bit microprocessor, the MPP PE is a *very* simple CPU indeed, but the large number of PEs result in MPP's tremendous performance. In Table CH.T1, MPPs operating speed on integer and floating point data is illustrated [Bat80].

The other components of MPP support the ARU. The Array Control Unit (ACU) has the following components:, the *main control* executes the main program and performs the necessary scalar arithmetic. It sends all array processing instructions to the *PE control,* which is responsible for broadcasting memory addresses and generating all ARU control signals. The *I/O control* is responsible for the shift operations of the S-registers and for interrupting regular processing for transfers between S-register and memory.

The *Program and Data Management Unit (PDMU)* is a DEC PDP-11 running the RSX-11M real-time operating system. It controls the overall program and data flow in the system, and is also used for program development.

The *Staging Memory* interfaces the PDMU and the external host to the ARU. Data arriving from the outside world, e.g., from an image sensor, will be organized as a stream of pixel values. The same is true for output, e.g., the output of a smoothed image to an image display device usually needs to be formatted as a stream of pixel values. The PEs on the other hand have to be loaded with a bit slice from all 128-by-128 pixel values at once. The staging memory (up to 40 Mbytes capacity) facilitates this reformatting process for the input and output data. For error recovery, single-bit error correction and double-bit error detection is provided.

The host machine is a DEC VAX-11/780. The VAX is connected to the PDMU and to the staging memory.

5.5. The STARAN Associative Processor

One of the first SIMD computers that has been built is the STARAN computer [FeF74, Dav74, Bat74, Bat76, Bat77]. The first STARAN was constructed during the early seventies by Goodyear Aerospace Corp.. STARAN is an SIMD machine, but is further classified as an associative processor. Associative processing was introduced in subsection 4.2 of this chapter. The heart of the machine consists of up to 32 associative arrays. Each array contains a bit-serial associative memory; 256 bit-serial processors; a mask register; a selector that selects as input the memory, the processors, the mask register, or an input channel; and a flip network that receives its input from the selector and can send its output to the memory, the processors, the mask register, or to an output channel (see Fig. CH.17).

The interconnection network used in each STARAN associative array is a 256 input, 256 output, eight-stage network called the flip network. Its topology is equivalent to that of the Generalized Cube network described in Chapter XX with each link being 1-bit wide. Its centralized control scheme is more limited than the approach discussed in that chapter. For further details of the flip network see [Bat76].

The memory is logically organized as a square array of 256-by-256 bits, and is constructed from commercial LSI memory chips. To facilitate easy access for associative processing, bit slice access mode is necessary; but since data arriving from the outside world is usually word organized (e.g., the complete personal record with name, salary, etc., from the example in subsection 4.2), word-slice access is also desirable. The STARAN memory does not only provide these two access modes but also combinations of the two, and is therefore called the multi-dimensional access *(MDA)* memory. The data area that is being fetched from memory is called *access stencil*. Any access stencil always fetches 256 bits. The form and position of the stencil are specified by two eight-bit parameters: the global address G and the stencil mode M. The logical location of a bit in memory is defined by the logical word location W and the logical bit location B. The processor to which a specific bit is routed is denoted by P. If data is not routed to or from the processors, P designates the bit position in the mask register or the output channel. If the processors access memory at global address G and with access mode M, the hardware for addressing and the network setting generated will cause processor P to read the bit from the following bit and word position:

$$B = (\overline{M}G) \oplus (MP)$$

$$W = (MG) \oplus (\overline{M}P).$$

For example, suppose M=0. Then the equations simplify to

$$B = (\bar{0}G) \oplus (0P) = G \oplus 0 = G$$

$$W = (0G) \oplus (\bar{0}P) = 0 \oplus P = P$$

Thus all processors receive bit-slice G; M=0 results in bit-slice access. On the other hand, if all bits of M are one, word-slice G is accessed:

$$B = (\bar{1}G) \oplus (1P) = 0 \oplus P = P$$

$$W = (1G) \oplus (\bar{1}P) = G \oplus 0 = G$$

Since M and G each consist of eight bits, 256 access modes are possible, as are 256 different positions for each access mode. The theory underlying this is described in [Bat77].

The associative arrays alone do not form the complete computer. Figure CH.18 shows a block diagram of the complete STARAN system. An associative processor (AP) control module contains all elements necessary to control the APs, e.g., instruction register, MDA address pointers, counters, etc. The parallel I/O (PIO Flip) module consists of a three-stage flip network with eight inputs and eight outputs, and is controlled by the PIO control. The flip network can permute data between its input ports, which are one 256-bit port for each AP module, and additional ports for disks and high-speed I/O. The Sequential Control is realized by a Digital Equipment Corporation PDP-11 and handles peripherals, interfaces to the system console, and performs diagnostic functions. The external function unit (EXF) synchronizes the AP, PIO, and sequential controllers. The Control Memory holds the instructions for the AP and PIO control. It is also connected to a host computer; thus STARAN can be considered a high-speed back end computer.

The use of STARAN has been studied for various application areas such as image processing, air traffic control, radar processing, and sonar processing. To give an example of the processing speed of STARAN, consider the fast Fourier transform (FFT). A system with four array modules can perform a 1024-point FFT on real data in approximately 3.0 milliseconds. An IBM 360/67 computer, a mainframe widely used when STARAN was designed, needs 446 milliseconds for the same task.

5.6. The Texas Reconfigurable Array Computer (TRAC)

TRAC, the Texas Reconfigurable Array Computer [SeU80, KaP80, PrK80, Lip84], is much more flexible than the previously described systems. Not only can it run in SIMD and in MIMD mode, but its processors can be combined to form processors with higher precision (e.g., four eight-bit processors can be combined to form a 32-bit processor). TRAC can therefore be reconfigured in a fashion that is most suitable for the current algorithm. For example, consider the edge tracing algorithm in Section CH.4. Not only can the machine run in SIMD mode and MIMD mode as was required by the algorithm, but in addition the best computational precision can be selected. The image in the example consisted of eight-bit integers; to do the smoothing and the global edge detection, 16-bit precision is necessary to take care of overflows. After these two operations, the picture consists of eight-bit values again. The calculation of the figure of merit might require the summation of all smoothed values in a subimage; thus, 24-bit or 32-bit integers may be needed if the subimage is big. Additional processors can be allocated to the task to form higher-precision computers. To perform the thresholding, eight-bit numbers have to be compared. No high precision results occur, so that the operation can be done by using eight-bit processors. Many of the processors necessary to calculate the figure of merit can therefore be released and reassigned to other tasks. The advantages of a system like TRAC become obvious from this discussion. The available processing power is always used in a very efficient manner. However, the flexibility of the machine increases the complexity of the system control.

The basic block diagram of TRAC is shown in Figure CH.19. It does not differ significantly from the Ultracomputer block diagram. Processors are connected to memory and I/O devices through the interconnection network. The powerful reconfiguration capabilities of TRAC are mainly hidden in the interconnection network.

The network is an SW-banyan network, an example of which is shown in Figure CH.20. A banyan network is represented by a graph with three types of nodes: the apex nodes associated with the processors, intermediate nodes representing switches, and base nodes associated with memory or I/O modules. The intermediate nodes all look alike; they all have the same *spread,* where the spread is the number of arcs going towards the apex nodes, and the same *fanout,* where fanout is the number of arcs going towards the base nodes. In Fig. CH.20, spread = fanout = 2, and the network is functionally equivalent to the generalized cube network in Chapter XX [Sie85, MaM81].

The network has two different operating modes, circuit-switched and packet-switched. The two modes are time multiplexed. This means that during one phase of the system clock the network is in the circuit-switched mode, and in the next clock phase it is used in packet-switched mode, resulting in two completely independent networks. The time multiplexing does not decreases the performance of the circuit-switched mode, since the packet-switching takes place while the memories execute their read or write cycle.

In SIMD mode, several processors receive the same instruction. In TRAC, this is implemented by an *instruction tree.* The instruction tree forms a path from one memory to the set of all processors that constitute the SIMD machine. If processors are combined to higher precision machines, a portion of the instruction tree is also used to propagate carry signals through the network. Each processor needs to get data from one or more memories of its own to facilitate the multiple data stream. This is accomplished by forming *data trees* that are independent of the instruction tree. Data trees are analogous to instruction trees except they go from the processor to one or more memories.

In MIMD mode, shared memory is desirable. A *shared memory tree* provides several processors with access to the same memory. Since the memories allow only one access at a given time, arbitration logic is needed that determines the next processor to access the shared memory; this logic uses a simple round-robin priority scheme. The dashed lines in Figure CH.20 could represent a shared memory tree.

The memories contain four Kbytes of storage each, and provide important additional features. Several index registers are associated with each memory module, which can be automatically incremented. This has an important implication for program and stack accesses. With the exception of branching conditions, instructions will be fetched consecutively from memory. Thus the processors do not need to send a memory address but just a request for the next memory word (=instruction); the index registers in the memory will provide the correct word and will be incremented in preparation for the next access. Stack operations that require increment or decrement of the index register after the memory access are simplified as well. Thus, instead of sending a 16-bit memory address through the network (which requires a 16-bit wide network path), only the appropriate index register in memory needs to be specified. This is done by using an 8-bit macro instruction, and the network path need only have an 8-bit width.

Secondary memory in TRAC is provided by self-managing secondary memory *(SMSM)* modules. The modules are segmented, and the segments can be accessed by name. The search for the segment name is done associatively to speed up accesses. SMSM connect to base nodes of the network.

A prototype of the system with four processors and nine memories, interconnected by a two-level banyan with a fanout of three and a spread of two, has been constructed.

5.7. The PASM Parallel Processing System

PASM is a multifunction partitionable SIMD/MIMD system being designed at Purdue University [SiS81, SiS84, KuS85c]. A 30-processor prototype is under construction [MeS85]. A block diagram of the basic components of PASM is shown in Figure CH.21. The *Parallel Computation Unit* (Figure CH.22) contains $N=2^n$ processors, N memory modules, and an interconnection network. The *processors* are microprocessors that perform the actual SIMD and MIMD operations. The *memory modules* are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. A memory module is connected to each processor to form a processor - memory pair called a *Processing Element (PE).* The N PEs are numbered from 0 to N−1. A pair of memory units is used for each module to allow data to be moved between one memory unit and secondary storage (the Memory Storage System), while the processor operates on data in the other memory unit. A PASM N=16 prototype will use Motorola MC68010 processors; the final N=1024 system, for which the architecture is designed, will employ sophisticated VLSI processors. The *interconnection network* provides a means for communication among the PEs. Two types of multistage interconnection networks are being considered for PASM: the Generalized Cube and the Augmented Data Manipulator. A fault tolerant variation of the Cube network, the Extra Stage Cube *(ESC)* network, is planned for inclusion in the prototype. An important property of the ESC network is its partitionability, i.e., the ability to divide the network into subnetworks that are completely independent of each other. These networks and their partitionabilities are described in the following chapter on Interconnection Networks.

The *Micro Controllers (MCs)* are a set of processors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. There are $Q=2^q$ MCs, physically addressed (numbered) from 0 to Q−1. Each MC controls N/Q PEs. PASM is being designed for Q=32 (Q=4 in the prototype). The MCs are multiple control units needed in order to have a partitionable SIMD/MIMD system. Each MC includes a memory module, which consists of a pair of memory units so that memory loading and computations can be overlapped. In SIMD mode, each MC fetches instructions and common data from its memory module, executes the control flow instructions (e.g., branches) and broadcasts the data processing instructions to its PEs. In MIMD mode, each MC gets instructions and common data for coordinating its PEs from its memory.

The partitioning rule in PASM requires all PEs in a partition of size 2^s to agree in their $n-s$ low-order bit positions. The physical addresses of the N/Q processors which are connected to an MC must all have the same low-order q bits so that they are in the same partition. The value of these low-order q bits is the physical address of the MC. A virtual SIMD machine of size RN/Q is obtained by having $R = 2^r$, $0 \leq r \leq q$, MCs use the same instructions and synchronizing the MCs. The physical addresses of the MCs must have the same low-order $q-r$ bits so that all PEs in the partition have the same low-order $q-r$ physical address bits. A virtual MIMD machine of size RN/Q is obtained similarly. In MIMD mode, the MCs may be used to help coordinate the activities of their PEs. Q is the maximum number of partitions allowable, and N/Q is the size of the smallest partition.

A masking scheme is used in SIMD mode for determining which PEs will be active, i.e., execute instructions broadcast to them by their MC. PASM will use PE address masks as described in subsection 3.3.

Control Storage contains the programs for the MCs. The loading of programs from Control Storage into the MC memory units is controlled by the System Control Unit.

The *Memory Storage System* provides secondary storage space for the Parallel Computation Unit for data files in SIMD mode, and for data and control instructions in

MIMD mode. It consists of N/Q independent *Memory Storage Units,* numbered from 0 to N/Q–1. Each Memory Storage Unit is connected to Q PE memory modules. For $0 \leq i \leq N/Q$, Memory Storage Unit i is connected to those PE memory modules whose physical addresses have the value i in their n–q high-order bits. Recall that, for $0 \leq k < Q$, MC k is connected to those PEs whose physical addresses have the value k in their q low-order bits. This is shown for N=32 and Q=4 in Figure CH.23.

Consider a virtual machine of RN/Q PEs, $R = 2^r$, $0 \leq r \leq q$, where the PEs are logically numbered from 0 to (RN/Q)–1, using the r+n–q high-order bits of their physical addresses as their logical addresses. To load such a virtual machine requires only R parallel block moves if the data for the PE memory module whose high-order n–q logical address bits equal i is loaded into Memory Storage Unit i. This is true no matter which group of R MCs (which agree in their low order q–r physical address bits) is chosen. As an example, consider Figure CH.23, and assume a virtual machine of size 16 is desired. The data for the PE memory modules whose logical addresses are 0 and 1 are loaded into Memory Storage Unit 0, for memory modules 2 and 3 into Unit 1, etc. Assume the partition of size 16 is chosen to consist of the PEs connected to MCs 1 and 3. The Memory Storage Units first simultaneously load PE memory modules physically addressed 1, 5, 9, 13, 17, 21, 25, and 19 (logically addressed 0, 2, 4, 6, 8, 10, 12, and 14), and then simultaneously load PE memory modules physically addressed 3, 7, 11, 15, 19, 23, 27, and 31 (logically addressed 1, 3, 5, 7, 9, 11, 13, and 15). No matter which pair of MCs is chosen (i.e., MCs 1 and 3, or MCs 0 and 2), only two parallel block loads are needed.

The *Memory Management System* controls the transferring of files between the Memory Storage System and the PEs. It is composed of a separate set of microprocessors (four in the prototype) dedicated to performing tasks in a distributed fashion. This distributed processing approach is chosen in order to provide the Memory Management System with a large amount of processing power at low cost. The division of tasks chosen is based on the main functions which the Memory Management System must perform.

The *System Control Unit* is responsible for the overall coordination of the other components of PASM. The types of tasks the System Control Unit will perform include program development, job scheduling, and coordination of the loading of the PE memory modules from the Memory Storage System with the loading of the MC memory modules from Control Storage. By carefully choosing which tasks should be assigned to the System Control Unit and which should be assigned to other system components (e.g., the MCs and Memory Management System), the System Control Unit can work effectively and not become a bottleneck. For the N=1024 PASM, the System Control Unit may consist of several processors in order to perform all of its functions efficiently. In the N=16 prototype, the System Control Unit is a microprocessor and the program development functions are performed by a host computer.

6. Comparison of the Various Parallel Processing Systems

Although the systems described in this chapter do not have that many similarities on first sight, an assessment of the various architectures and some general comments are appropriate. Cm* is the only system of the ones surveyed that employs a shared bus scheme. All other systems use an interconnection network. The complexity of this network ranges from the simple nearest-neighbor approach employed in the MPP to more flexible multistage networks as in STARAN, Ultracomputer, TRAC, and PASM. STARAN uses a centralized control scheme with which multiple network switches are set by the same control signal. In the Ultracomputer, TRAC, and PASM, each switch is controlled individually. The Ultracomputer adds hardware for handling F&A

primitives, while TRAC adds hardware for implementing both circuit and packet switching.

Another aspect is the PE-to-PE and the processor-to-memory organization. MPP and PASM are PE-to-PE and TRAC is processor-to-memory. The Ultracomputer is basically a processor-to-memory organization, but since a cache is associated with the processor, the organization becomes hybrid. Machines with a PE-to-PE organization sacrifice the advantage of shared memory. Shared memory is desirable for many parallel processing applications where processors need to access the same data. On the other hand, in a PE-to-PE system local memory accesses are not routed through a network, avoiding potentially sizeable delay for every memory access. Thus, problems which require little communication between PEs can be processes faster.

The problem domain for which the various systems were designed was important for their basic architecture decision. STARAN, TRAC, Cm*, and the Ultracomputer are general purpose machines with no preferred problem domain. The design of MPP and PASM was strongly guided by image processing applications. They will perform image processing especially well, though they are also suited for other parallel computations. For image processing, large amounts of data have to be brought into the machine at high speed. This resulted in the staging memory for MPP and the complex secondary memory system for PASM. Since most calculations are performed on data in the local memory or in neighboring PEs, a simple interconnection network was chosen for MPP. The Ultracomputer and TRAC can have I/O units that replace some of the memories. TRAC is the only machine of those surveyed that can combine simple processors to achieve higher precision. MPP and STARAN are bit-serial, while TRAC, PASM, and the Ultracomputer use complex processors of 16 or more bits.

A final distinction that will be made here is based on reconfigurability. MPP and STARAN are reconfigurable in a very limited sense: their networks can be reconfigured, and their PEs can be enabled and disabled. They will work in SIMD mode only. Cm* and the Ultracomputer are pure MIMD machines. Since each processor in an MIMD system follows its individual instruction stream, both machines can form one or more virtual MIMD submachines. In Cm*, small virtual machines should not be intercluster since communications along the intercluster busses is slow; the partitions in the Ultracomputer are arbitrary due to its flexible network. PASM and TRAC are both reconfigurable systems since they can work in both the MIMD and the SIMD mode of operation, and can, in addition, be partitioned into virtual machines.

Tradeoffs of the different approaches are application dependent. Construction and use of the systems will enhance the knowledge of what parallel system features are important for different types of computation.

7. Problems

CH.1 a) Consider the CAM algorithm given in Figure CH.11. Is it possible to exchange the ordering of the two "select CAM cell" operations and their associates "writes"? Why or why not?

b) Write a CAM algorithm that subtracts one from an integer in all CAM cells.

CH.2 Consider a hypothetical Cm* system with the following parameters:

10 computer modules per cluster

5 clusters

2 intercluster busses

Both the intercluster busses serve all of the 5 clusters. It is found that the average computer module generates 1 million memory references per second; 90% of these are local references, 7% are intracluster, and 3% are intercluster. Local busses (which connect the memory to the Slocal), MAP busses, and intercluster busses can all achieve a maximum throughput rate of two million references per second (where a reference consists of both a request and the reply). A reference appearing on an intercluster bus goes to the different clusters with equal likelihood; within a cluster, a reference goes to the different computer modules with equal likelihood.

(a) What is the local bus utilization (actual number of references / maximum references possible)?

(b) What is the MAP bus utilization?

(c) What is the intercluster bus utilization?

CH.3 a) In the Ultracomputer, suppose the network supports only F&A operations. How can LOAD and STORE operations be implemented with this network?

b) Recall that network nodes in the Ultracomputer can combine requests. Describe how a network node can combine LOAD and LOAD, LOAD and STORE, and STORE and STORE operations.

CH.4 In MPP, assume a task consists of loading an eight-bit/pixel 128-by-128 pixel image into memory, perform some operation on it, and move the 8-bit/pixel result out of the machine. This task is to be repeated for 200 images. Describe how load, unload, and execution are overlapped. Determine the maximum number of 100 ns cycles that can be used for execution on any one image so that loading and unloading take place at maximum speed.

CH.5 The access stencil size and location in the STARAN associative processor is determined by formulas given in the chapter. Determine which memory locations are accessed by M = (00000111) and G = (10000000).

CH.6 Consider the PASM system. Assume not N/Q but $(N/Q)/2^d$ distinct Memory Storage Units are available. How many parallel block loads are required to load a virtual machine of size RN/Q? Describe the MSU-to-PE connections. Discuss data allocation.

CH.7 Comment on the various problems associated with using a general purpose, commercially available microprocessor (e.g., Intel 286, Motorola 68010, etc.) as the basis for a PE in an SIMD machine. Consider how instructions are to be broadcast to each PE from the control unit and the difficulties associated with the microprocessor instruction prefetch mechanism. Draw a block diagram of the external hardware required to facilitate instruction broadcast as well as to ensure synchronization among PEs. Describe how processor enabling/disabling could be implemented.

CH.8 As illustrated in this chapter, two distinctly different approaches toward implementation of parallel processing systems are use of simple bit-serial PEs (e.g., STARAN and MPP) and the use of complex microprocessor-based PEs (e.g., PASM). Discuss the tradeoffs between these two approaches. Consider such factors as I/O requirements, interconnection complexity, ease of custom VLSI PE design, and amenability to intended applications (e.g., image processing, weather forecasting, etc.).

CH.9 Contrast various problems associated with implicit extraction of parallelism from commonly-used high level languages (C, Pascal, Fortran, Ada) with those associated with explicit specification of parallelism using "parallel extensions" of these same languages. Consider such factors as program portability, compiler portability, optimization of compiled code, and ease of programmability.

8. References

[BaB68] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Transactions on Computers,* Vol. C-17, August 1968, pp. 746-757.

[Bae80] J-L. Baer, *Computer Systems Architecture,* Computer Science Press, Potomac, MD, 1980.

[Bat74] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS 1974 National Computer Conference,* May 1974, pp. 405-410.

[Bat76] K. E. Batcher, "The flip network in STARAN," *1976 International Conference on Parallel Processing,* August 1976, pp. 65-71.

[Bat77] K. E. Batcher, "The multidimensional access memory in STARAN," *IEEE Transactions on Computers,* Vol. C-26, February 1977, pp. 174-177.

[Bat80] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers,* Vol. C-29, September 1980, pp. 836-844.

[Bat82] K. E. Batcher, "Bit serial parallel processing systems," *IEEE Transactions on Computers,* Vol. C-31, May 1982, pp. 377-384.

[BeD85] J. Beetem, M. Denneau, and D. Weingarten, "The GF11 supercomputer," *12th Annual International Symposium on Computer Architecture,* June 1985, pp. 108-115.

[BoD72] W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proceedings of the IEEE,* Vol. 60, April 1972, pp. 369-388.

[BrF79] F. A. Briggs, K-S. Fu, K. Hwang, and J. H. Patel, "PM4 - a reconfigurable multimicroprocessor system for pattern recognition and image processing," *AFIPS 1979 National Computer Conference,* June 1979, pp. 255-265.

[ClS84] C. Cline and H. J. Siegel, "A comparison of parallel language approaches to data representation and data transferral," *Computer Data Engineering Conference (COMPDEC),* April 1984, pp. 60-66.

[ClS85] C. L. Cline and H. J. Siegel, "Augmenting Ada for SIMD parallel processing," *IEEE Transactions on Software Engineering,* Vol. SE-11, September 1985, pp. 970-977.

[Con63] M. E. Conway, "A multiprocessor system design," *AFIPS 1963 Fall Joint Computer Conference,* 1963, pp. 139-146.

[CrG85] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing,* August 1985, pp. 531-540.

[Dav74] E. W. Davis, "STARAN parallel processor system software," *AFIPS 1974 National Computer Conference,* May 1974, pp. 17-22.

[Dij68] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages,* F. Genuys, ed., Academic Press, New York, NY, 1968, pp. 43-112.

[DuH73] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis,* John Wiley and Sons, New York, NY, 1973.

[EdG85] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *12th International Symposium on Computer Architecture,* June 1985, pp. 126-135.

[FeF74] J. D. Feldman and L. C. Fulmer, "RADCAP--an operational parallel processing facility," *AFIPS 1974 National Computer Conference,* May 1974, pp. 7-15.

[Fly66] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[Fos76] C. A. Foster, *Content Addressable Parallel Processors,* Van Nostrand, Reinhold, New York, NY, 1976.

[Fre61] H. Freeman, "Techniques for the digital computer analysis of chain-encoded arbitrary plane curves," *Proc. NEC,* Vol. 17, October 1961, pp. 421-432.

[GoG83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing an MIMD shared-memory parallel computer," *IEEE Transactions on Computers,* Vol. C-32, February 1983, pp. 175-189.

[HaD85] M. T. Hagan, H. B. Demuth, and P.H. Singgih, "Parallel signal processing on the HEP," *1985 International Conference on Parallel Processing,* August 1985, pp. 599-606.

[Hab76] A. N. Habermann, *Introduction to Operating System Design,* SRA, Chicago, IL, 1976.

[HoJ81] R. W. Hockney and C. R. Jeshope, *Parallel Computers,* Adam Hilger Ltd., Bristol, England, 1981.

[HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, NY, 1984.

[JaM85] L. H. Jamieson, P. T. Mueller, and H. J. Siegel, "FFT algorithms for SIMD parallel processing systems," *Journal of Parallel and Distributed Computing,* to appear.

[JoC77] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans, "Software management of Cm* - a distributed multiprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 657-663.

[KaK79] S. I. Kartashev and S. P. Kartashev, "A multicomputer system with dynamic architecture," *IEEE Transactions on Computers,* Vol. C-28, October 1979, pp. 704-720.

[KaK82] S. P. Kartashev and S. I. Kartashev, "Distribution of programs for a system with dynam'c architecture," *IEEE Transactions on Computers,* Vol. C-31, June 1982, pp. 488-514.

[KaP80] R. N. Kapur, U. V. Premkumar, and G. J. Lipovski, "Organization of the TRAC processor-memory subsystem," *AFIPS 1980 National Computer Conference,* June 1980, pp. 623-629.

[KuF85] J. T. Kuehn, J. A. Fessler, and H. J. Siegel, "Parallel image thinning and vectorization on PASM," *1985 IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* June 1985, pp. 368-374.

[KuS85a] J. T. Kuehn, H. J. Siegel, D. L. Tuomenoksa, and G. B. Adams III, "The use and design of PASM," in *Integrated Technology for Parallel Image Processing,* S. Levialdi, ed., Academic Press, San Diego, CA, 1985, pp. 133-152.

[KuS85b] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," *1985 International Conference on Parallel Processing,* August 1985, pp. 232-235.

[KuS85c] J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems,* to appear, December 1985.

[LaL75] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randall, "Glypnir-a programming language for Illiac IV," *Communications of the ACM,* Vol. 18, March 1975, pp. 157-164.

[Lip84] G. J. Lipovski, "The Banyan switch in TRAC the Texas Reconfigurable Array Computer," *Distributed Processing Technical Committee Newsletter (IEEE Computer Society),* January 1984, pp. 13-26.

[MaM81] M. Malek and W. W. Myre, "A description method of interconnection networks," *IEEE Technical Committee on Distributed Processing Quarterly,* February 1981, pp. 1-6.

[MeS85] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Computer Society Spring Compcon 85,* February 1985, pp. 429-434.

[MiR81] O. R. Mitchell, A. P. Reeves, and K-S. Fu, "Shape and texture measurements for automated cartography," *1981 IEEE Computer Society Conference on Pattern Recognition and Image Processing,* August 1981, pp. 367.

[MuS80] P. T. Mueller, Jr., L. J. Siegel, and H. J. Siegel, "A parallel language for image and speech processing," *IEEE Computer Society Fourth International Computer Software and Applications Conference,* October 1980, pp. 476-483.

[Nut77a] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Fourth Annual Symposium on Computer Architecture,* March 1977, pp. 147-152.

[Nut77b] G. J. Nutt, "A parallel processor operating system comparison," *IEEE Transactions on Software Engineering,* Vol. SE-3, November 1977, pp. 467-475.

[Per79] R. H. Perrott, "A language for array and vector processors," *ACM Transactions on Programming Languages and Systems,* Vol. 1, October 1979, pp. 177-195.

[PfB85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," *1985 International Conference on Parallel Processing,* August 1985, pp. 764-771.

[PrK80] U. V. Premkumar, R. N. Kapur, M. Malek, G. J. Lipovski, and P. Horne, "Design and implementation of the banyan interconnection network in TRAC," *AFIPS 1980 National Computer Conference,* June 1980, pp. 643-653.

[ReB80] A. P. Reeves and J. D. Bruner, "The programming language Parallel Pascal," *1980 International Conference on Parallel Processing,* August 1980, pp. 5-7.

[Red73] S. F. Reddaway, "DAP - A distributed array processor," *1st Annual Symposium on Computer Architecture,* December 1973, pp. 61-65.

[SeU80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 National Computer Conference,* June 1980, pp. 631-641.

[Sei85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM,* January 1985, pp. 22-33.

[SiS81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers,* Vol. C-30, December 1981, pp. 934-947.

[SiS82] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," *IEEE Transactions on Computers,* Vol. C-31, March 1982, pp. 208-218.

[SiS84] H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: a reconfigurable parallel system for image processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition,* July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter: *Computer Architecture News,* Vol. 12, No. 4, September 1984, pp. 7-19).

[Sic77] H. J. Siegel, "Controlling the active/inactive status of SIMD machine processors," *1977 International Conference on Parallel Processing,* August 1977, pp. 183.

[Sie85] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, Lexington, MA, 1985.

[SnJ85] L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel (Editors), *Algorithmically Specialized Parallel Computers,* Academic Press, Orlando, FL, 1985.

[Ste75] K. G. Stevens, "CFD - A FORTRAN-like language for the Illiac IV," *ACM Conference on Programming Languages and Compilers for Parallel and Vector Machines,* March 1975, pp. 72-76.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture (second edition),* H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[SwB77] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout, "The implementation of the Cm* multimicroprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 645-655.

[SwF77] R. J. Swan, S. Fuller, and D. P. Siewiorek, "Cm*: a modular multimicroprocessor," *AFIPS 1977 National Computer Conference,* June 1977, pp. 637-644.

[TuA83] D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *1983 IEEE Computer Society Symposium on Computer Vision and Pattern Recognition,* June 1983, pp. 336-344.

[Uhr81] L. Uhr, "A language for parallel processing of arrays, embedded in PASCAL," in *Languages and Architectures for Image Processing,* M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 53-88.

[WaS82] M. R. Warpenburg and L. J. Siegel, "SIMD image resampling," *IEEE Transactions on Computers,* Vol. C-31, October 1982, pp. 934-942.

[YoJ85] M. A. Yoder and L. H. Jamieson, "Simulation of a word recognition system on two parallel architectures," *1985 International Conference on Parallel Processing,* August 1985, pp. 171-179.

[YoS82] M. A. Yoder and L. J. Siegel, "Dynamic time warping algorithms for SIMD machines and VLSI processor arrays," *1982 International Conference on Acoustics, Speech, and Signal Processing,* May 1982, pp. 1274-1277.

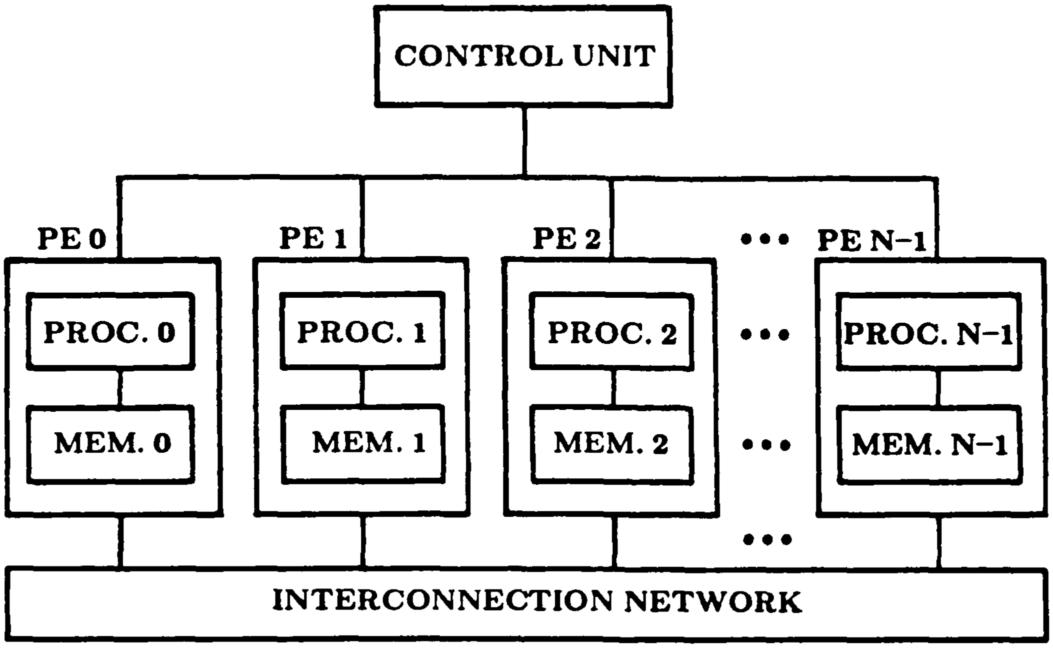


Figure CH.1 Processing-Element-to-Processing-Element SIMD machine configuration with N processing elements (PEs).

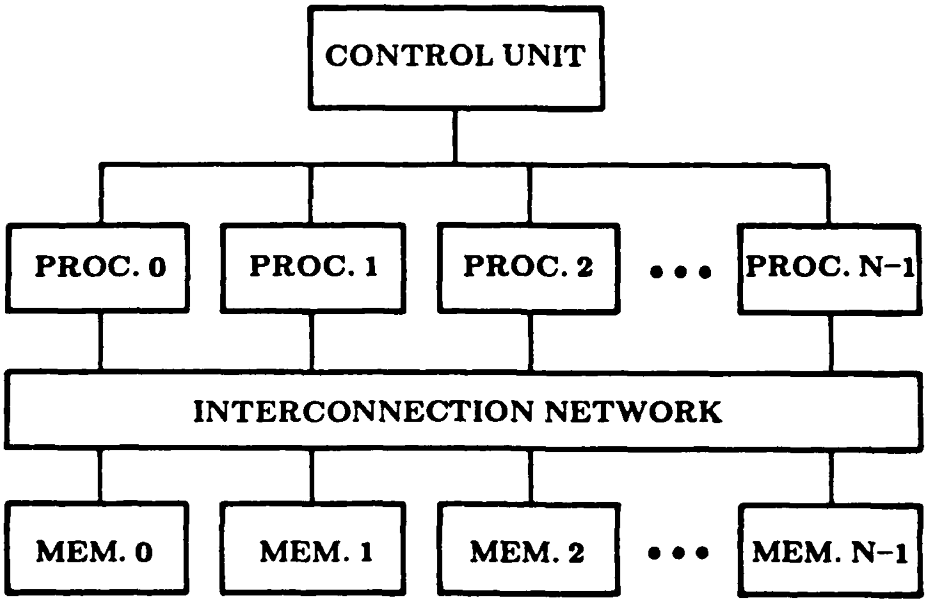


Figure CH.2 Processor-to-Memory SIMD machine configuration with N processors and N memories.

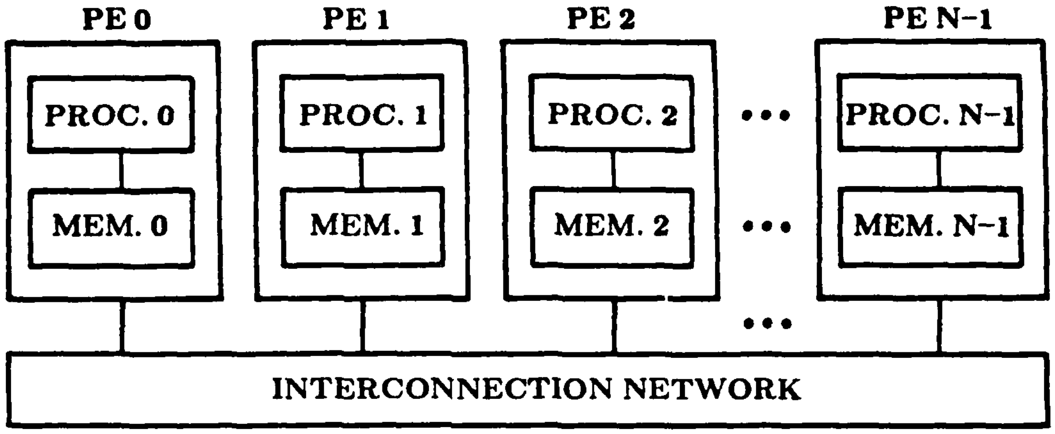


Figure CH.3 Processing-Element-to-Processing-Element MIMD machine configuration with N processing elements (PEs).

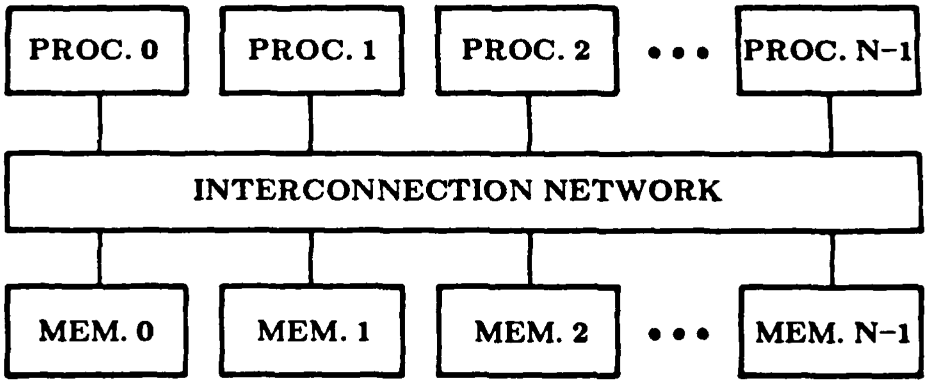


Figure CH.4 Processor-to-Memory MIMD machine configuration with N processors and N memories.

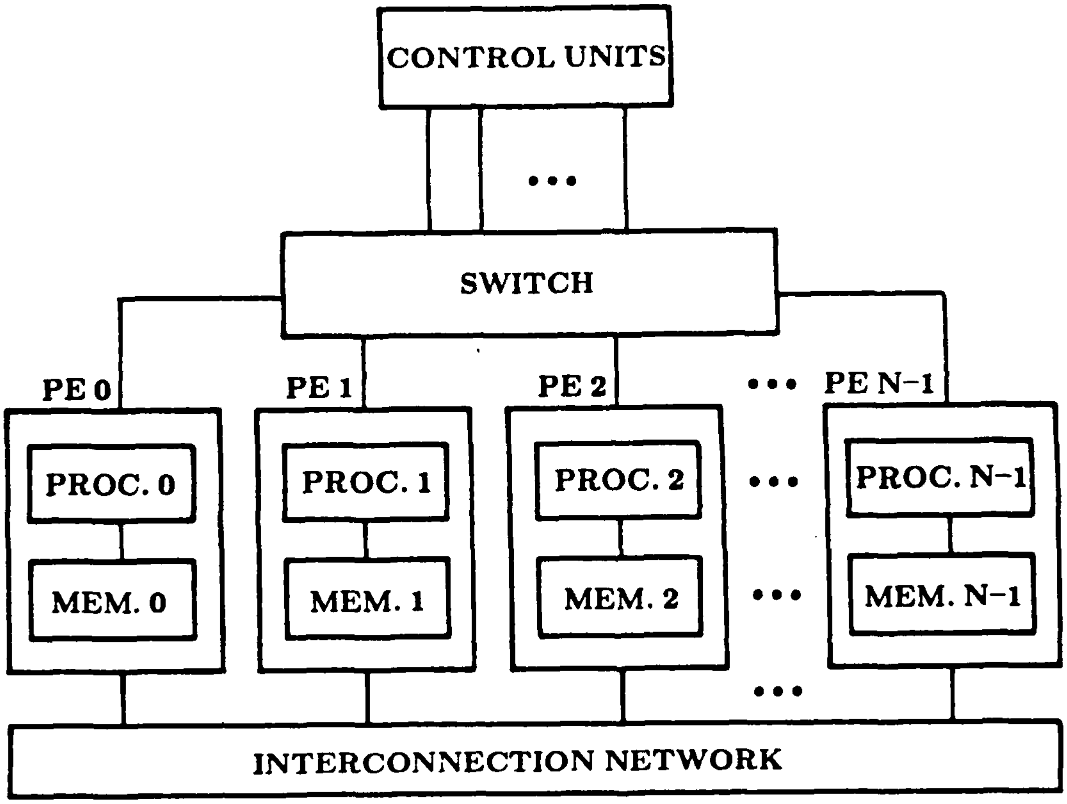


Figure CH.5 General model of a Multiple-SIMD system.

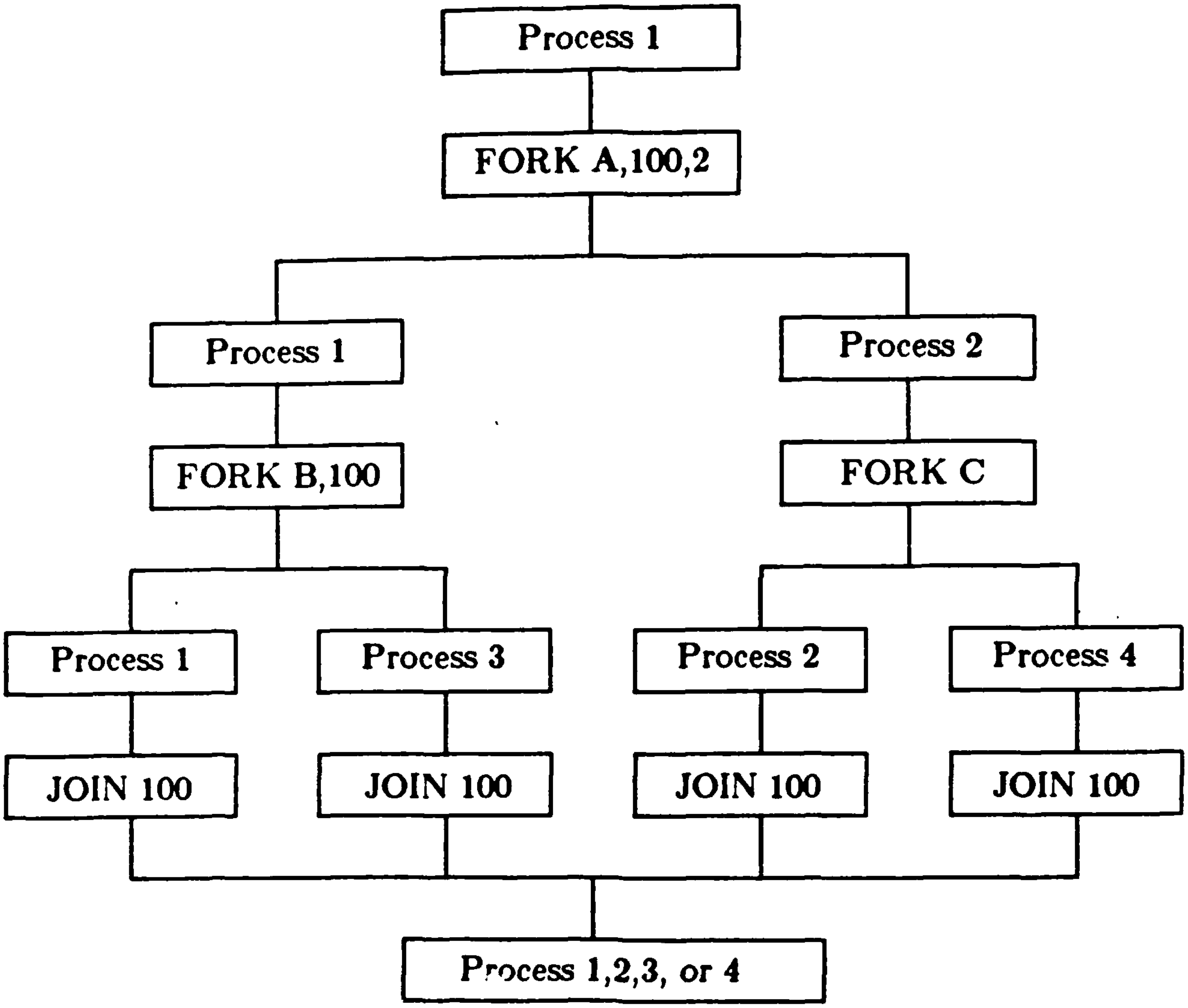


Figure CH.6 Flowchart of a parallel program using FORK and JOIN statements.

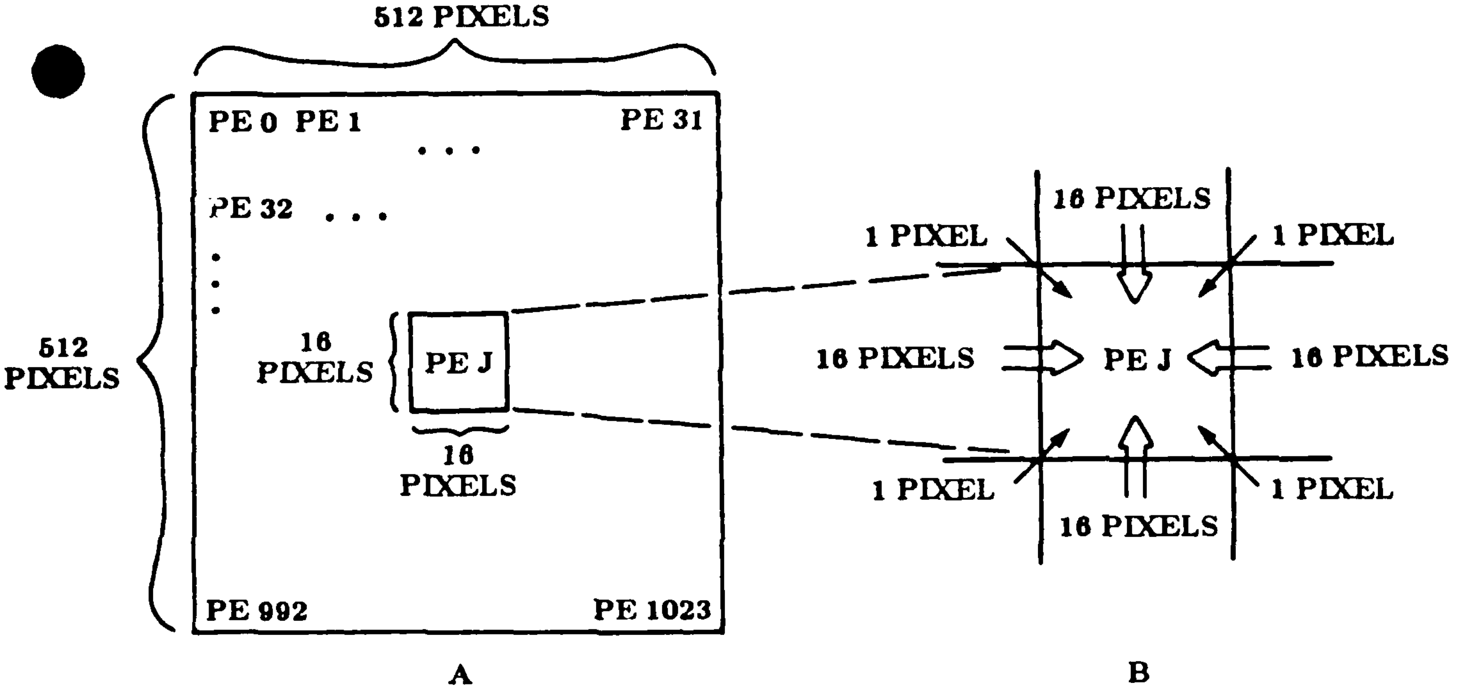


Figure CH.7 Data allocation and inter-PE transfers for the image smoothing algorithm.

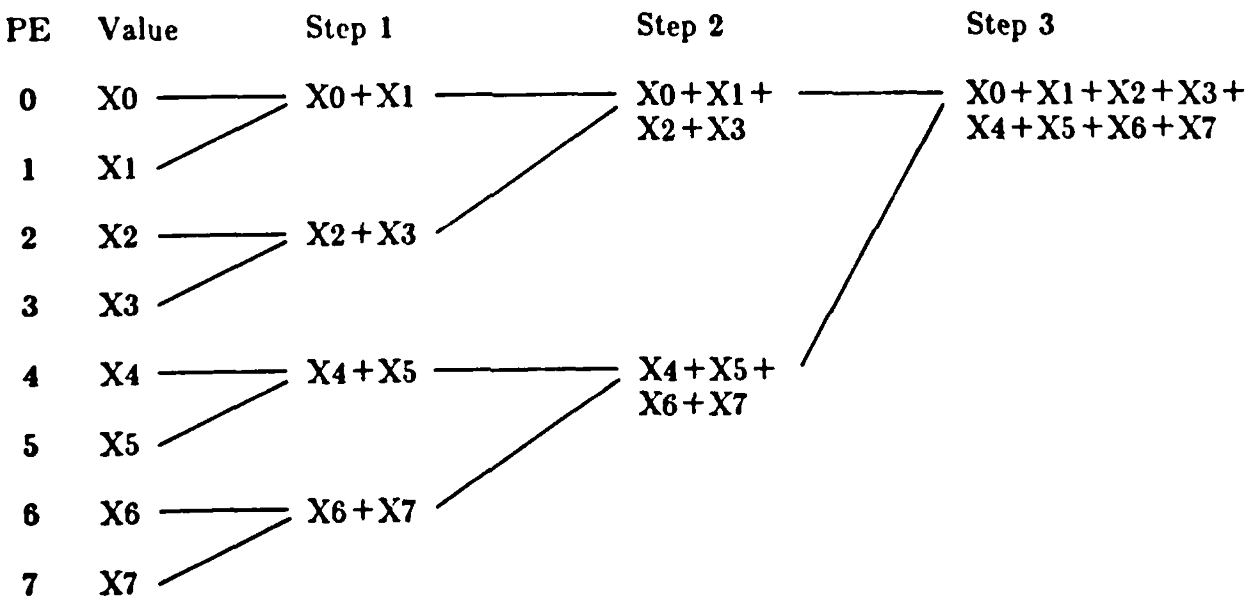


Figure CH.8 Recursive doubling with eight PEs.

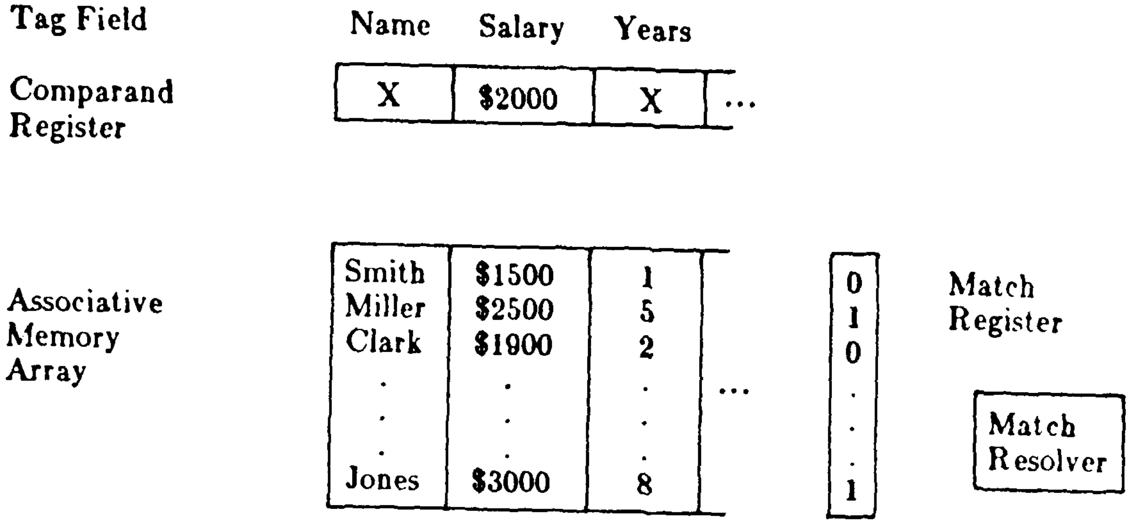


Figure CH.9 Basic operation of a content addressable memory.

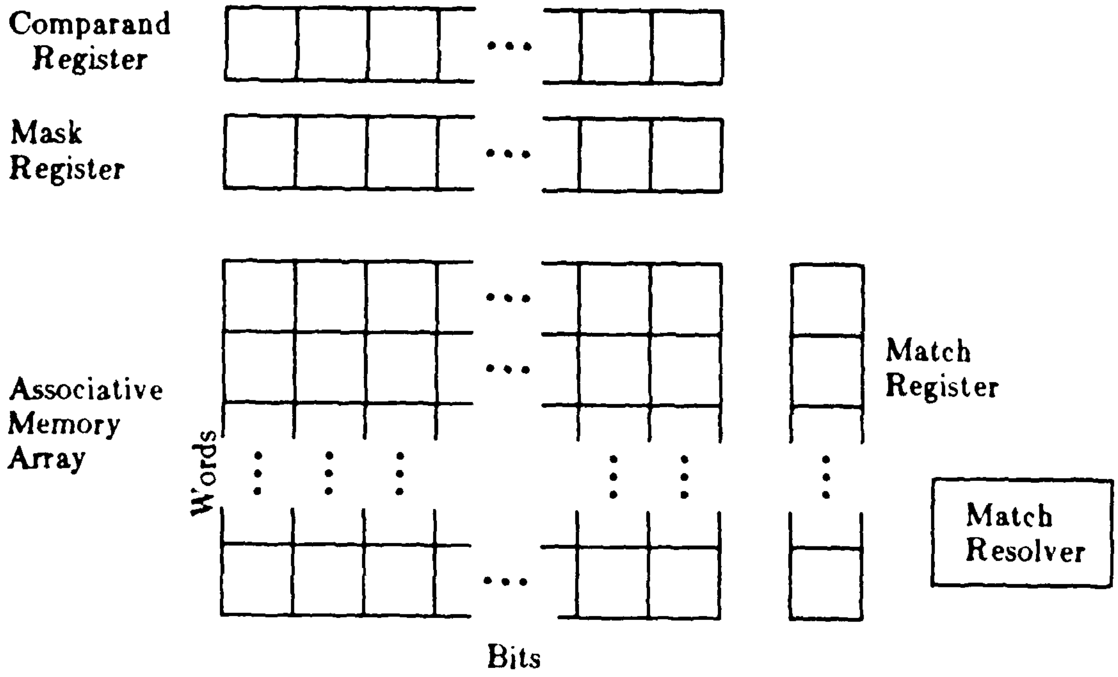


Figure CH.10 Hardware organization of a content addressable memory.

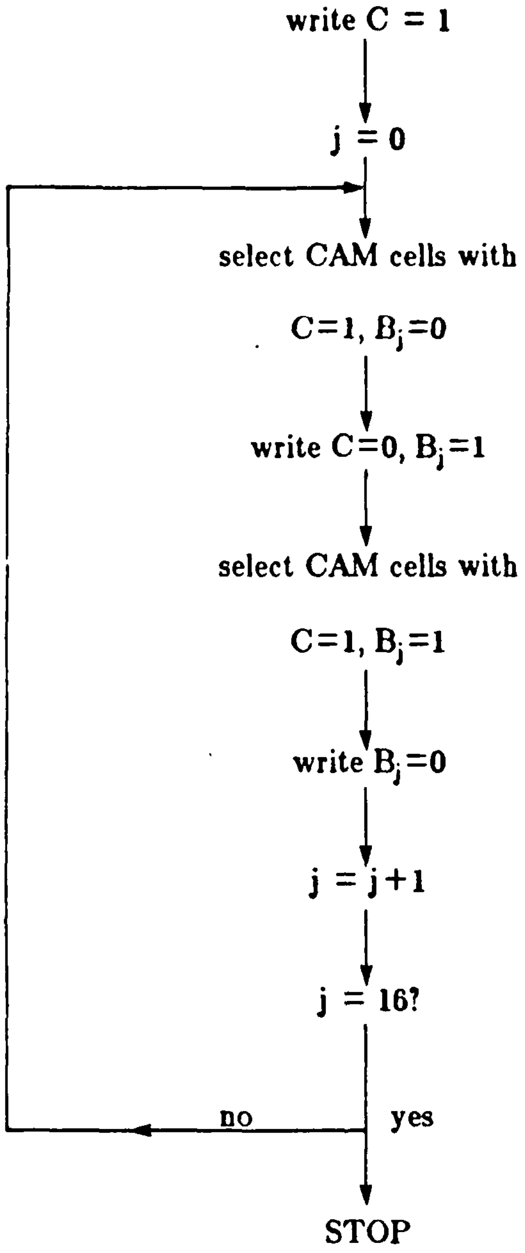


Figure CH.11 CAM algorithm to add one to an integer in all CAM cells.

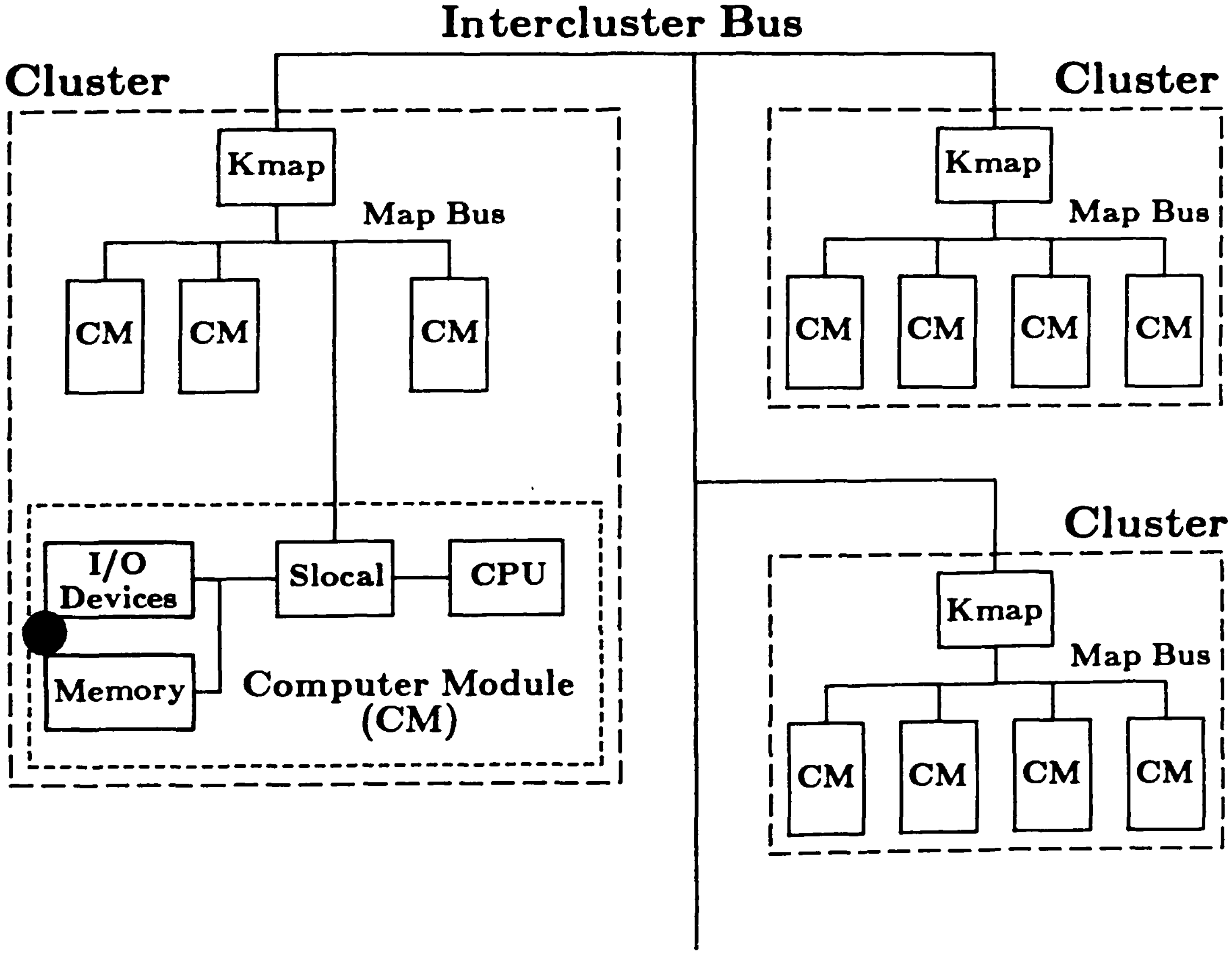


Figure CH.12 Block diagram of a simple 3 cluster Cm* system.

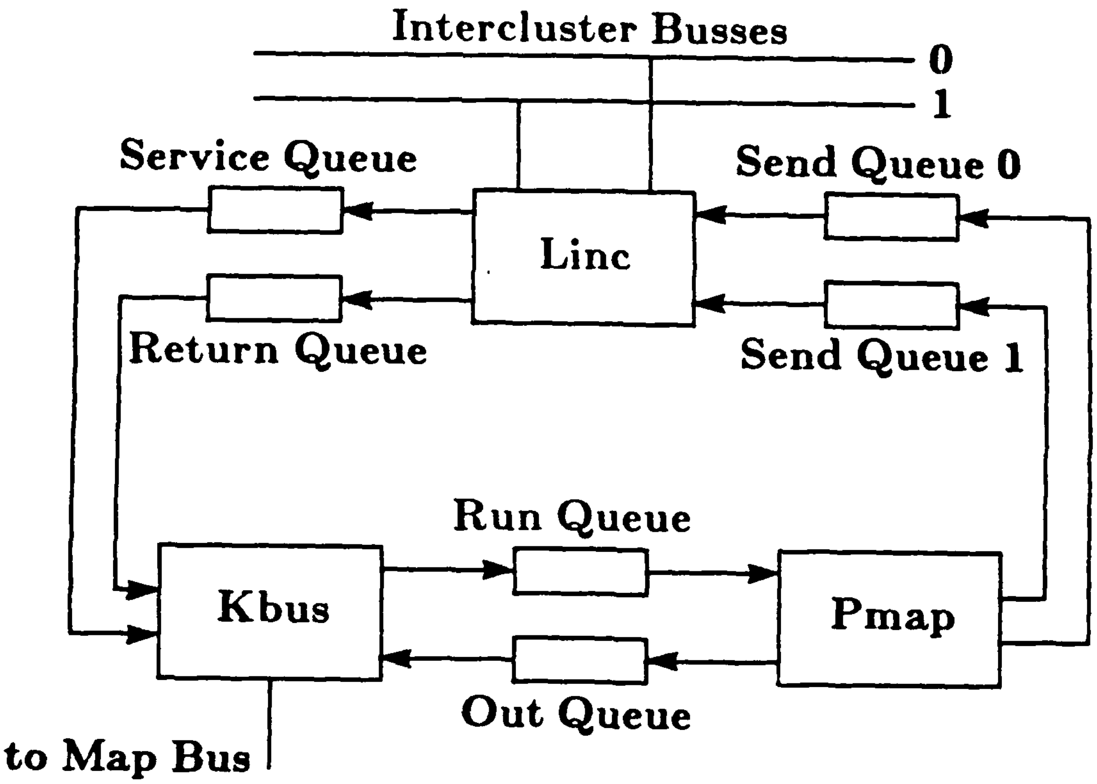


Figure CH.13 Block diagram of the Kmap of Cm*.

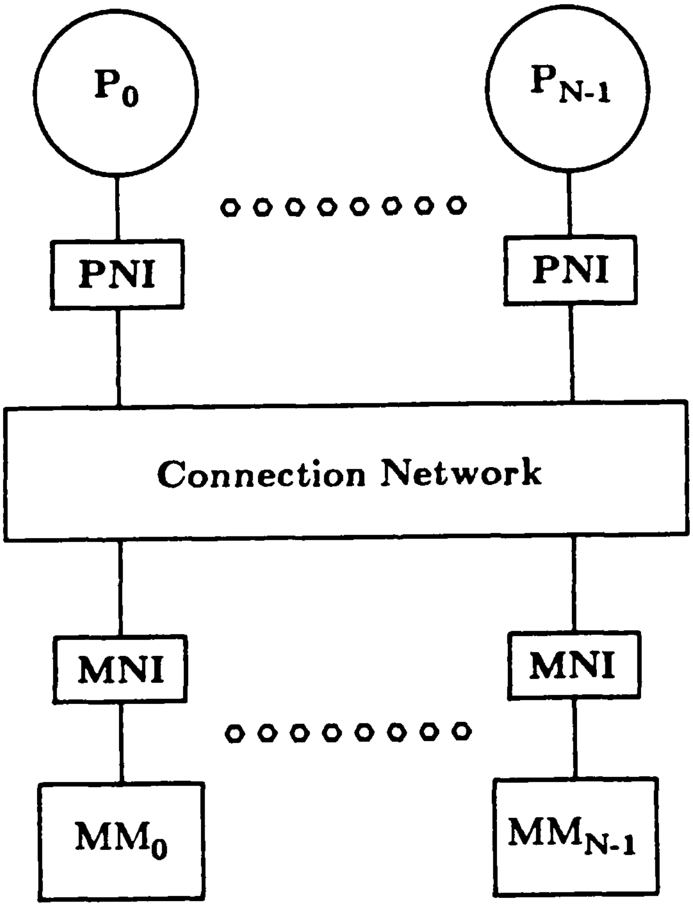


Figure CH.14 Block diagram of the NYU Ultracomputer. © 1983 IEEE [GoG83].

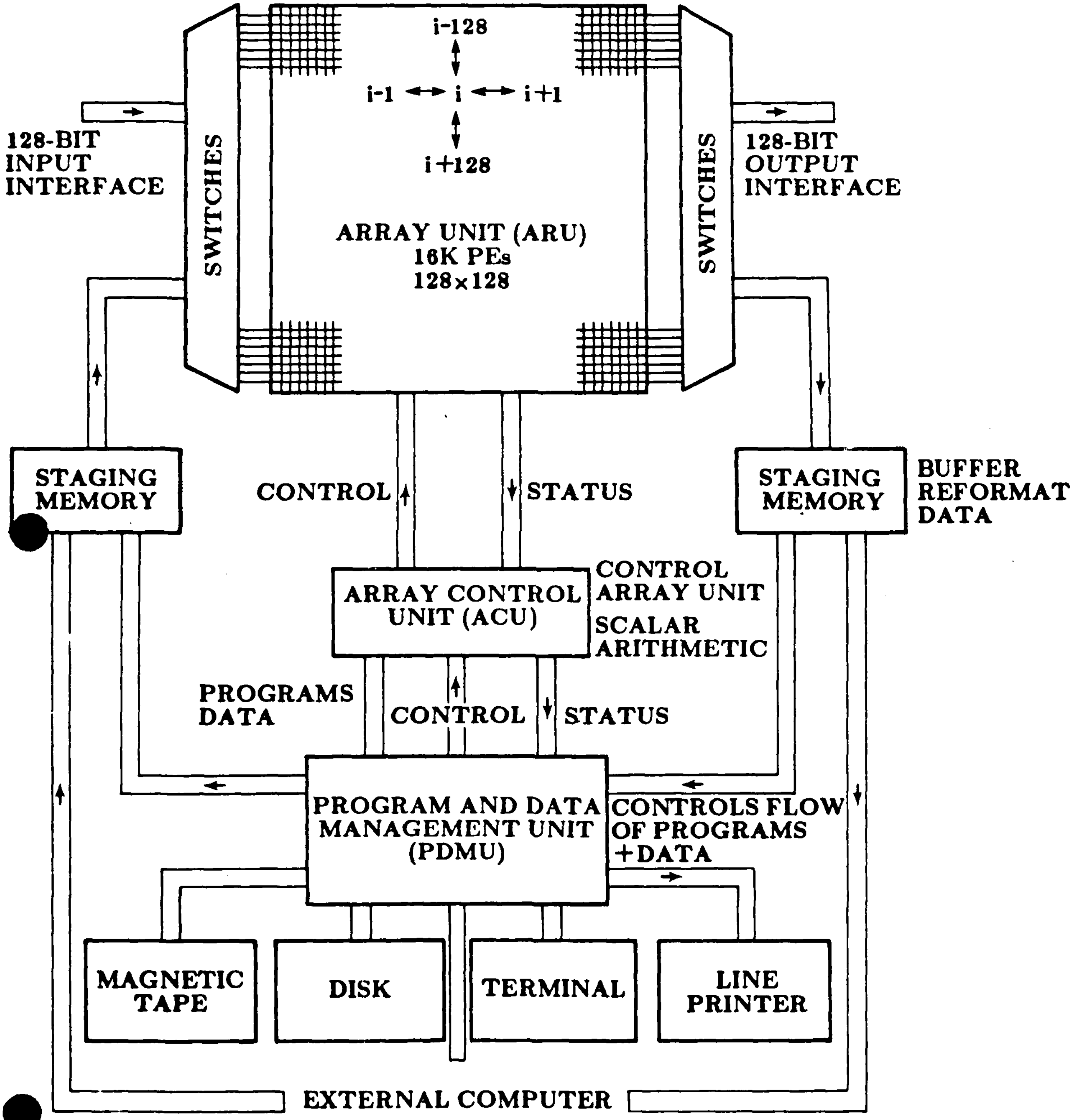


Figure CH.15 Block diagram of thc MPP. © 1982 IEEE [Bat82].

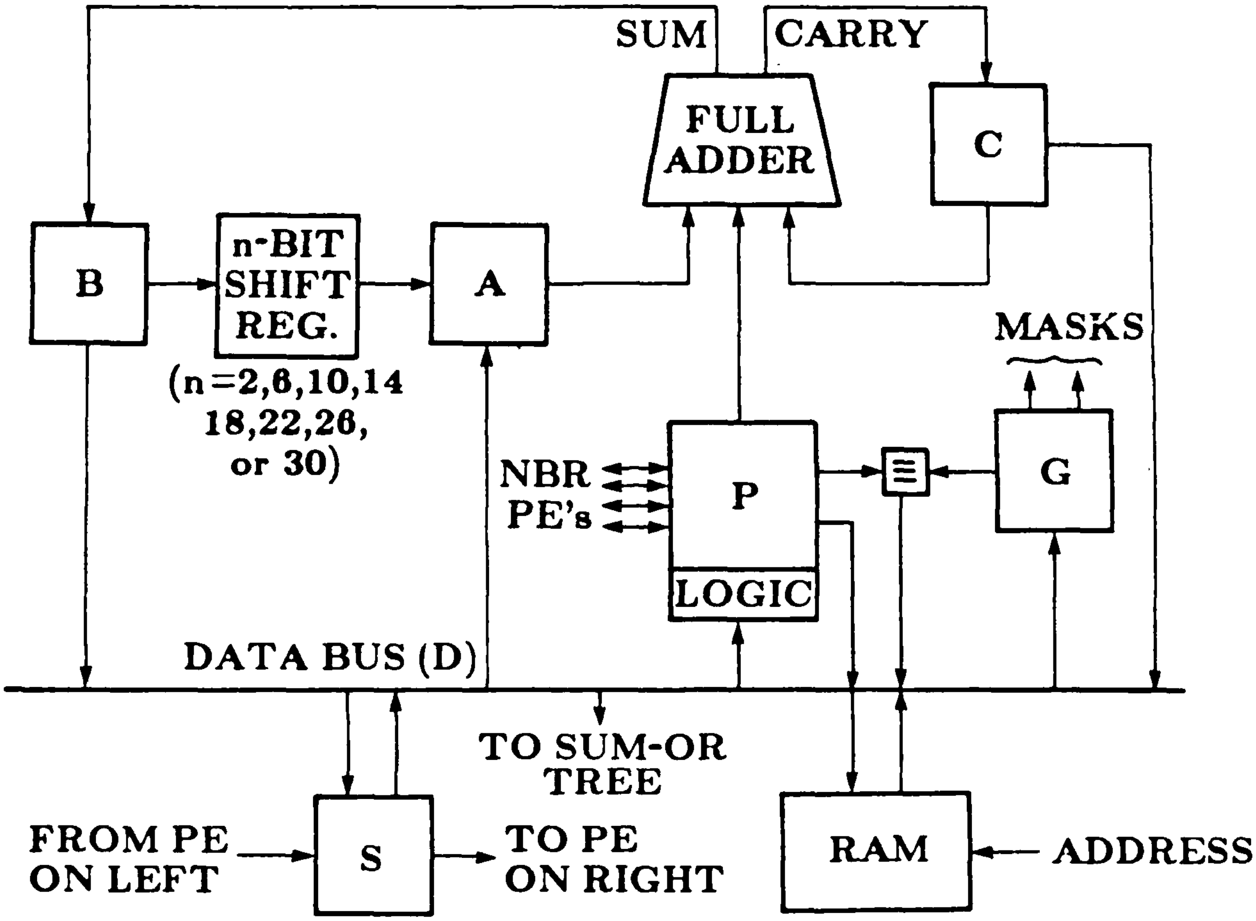


Figure CH.16 Block diagram of an MPP Processing Element. © 1982 IEEE [Bat82].

Table CH.1 Processing speed of MPP. © 1982 IEEE [Bat82].

Operations	Execution Speed*
Addition of Arrays	
8-bit integers (9-bit sum)	6553
12-bit integers (13-bit sum)	4128
32-bit floating-point numbers	430
Multiplication of Arrays (Element-by-Element)	
8-bit integers (16-bit product)	1861
12-bit integers (24-bit product)	910
32-bit floating-point numbers	216
Multiplication of Array by Scalar	
8-bit integers (16-bit product)	2340
12-bit integers (24-bit product)	1260
32-bit floating-point numbers	373

*Million Operations per Second (MOPS)

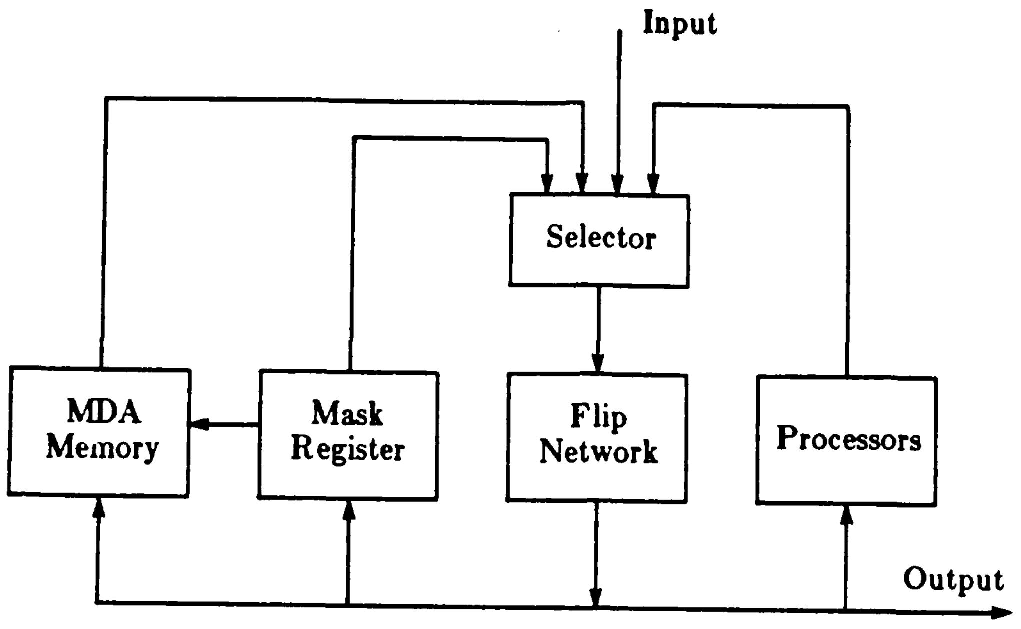


Figure CH.17 Block diagram of the STARAN associative array.

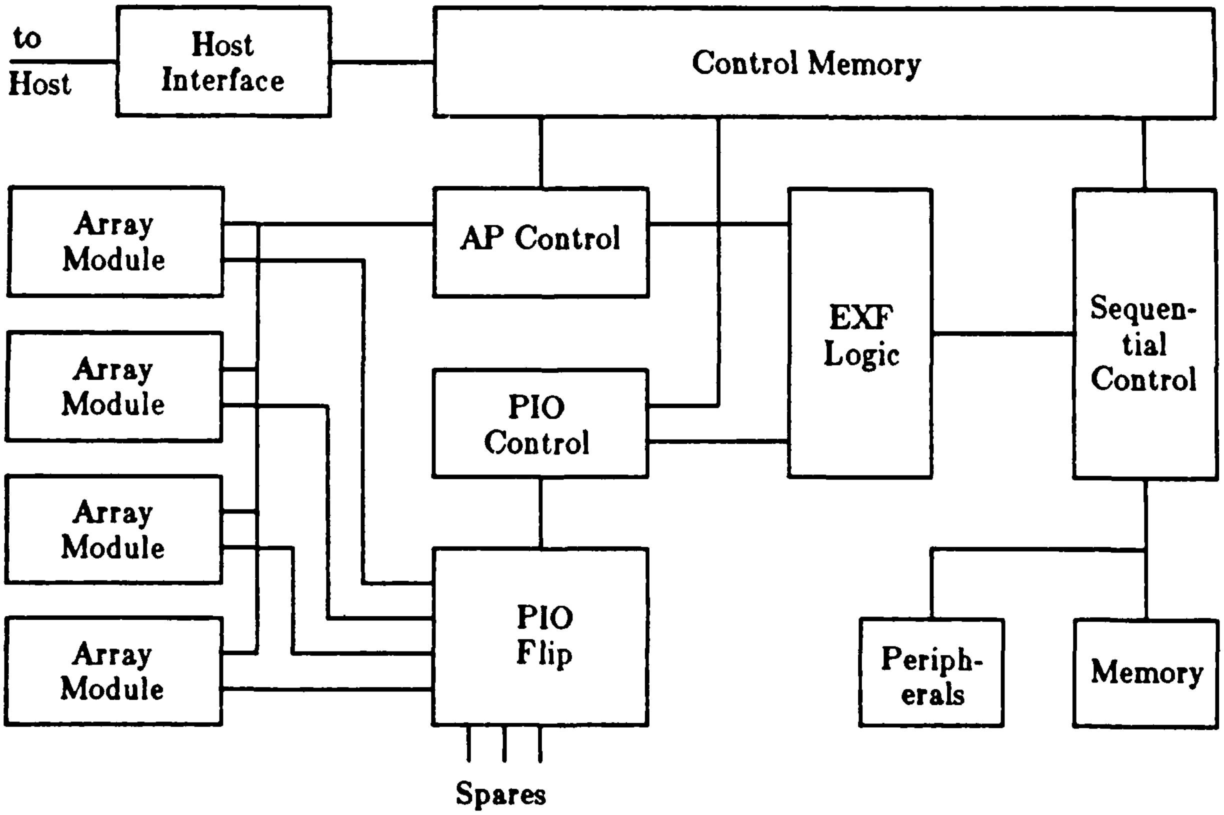


Figure CH.18 Block diagram of the complete STARAN system.

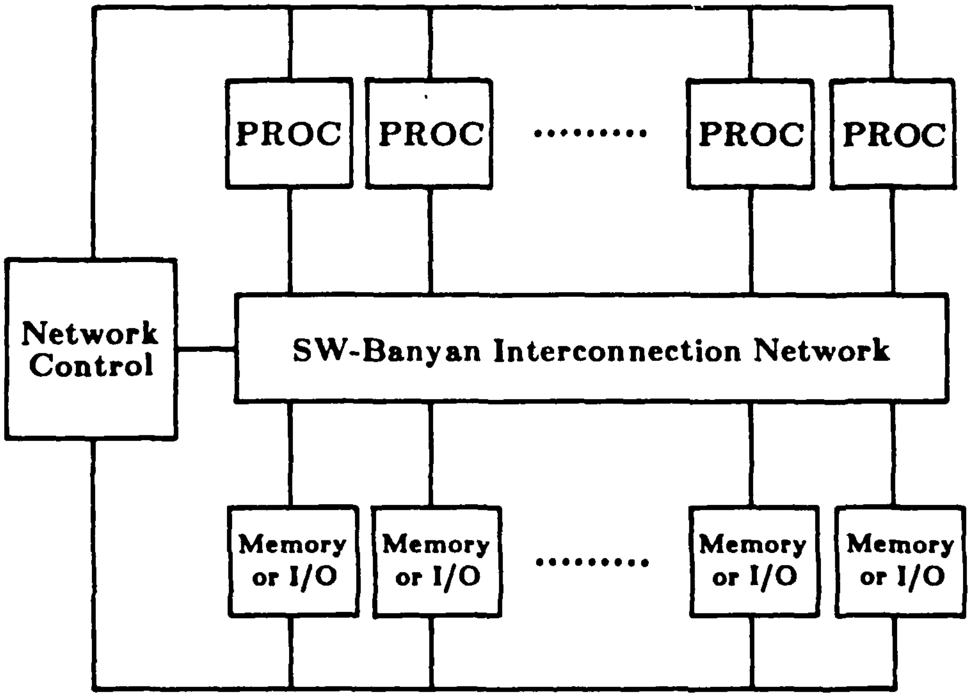


Figure CH.19 Block diagram of the Texas Reconfigurable Array Computer.

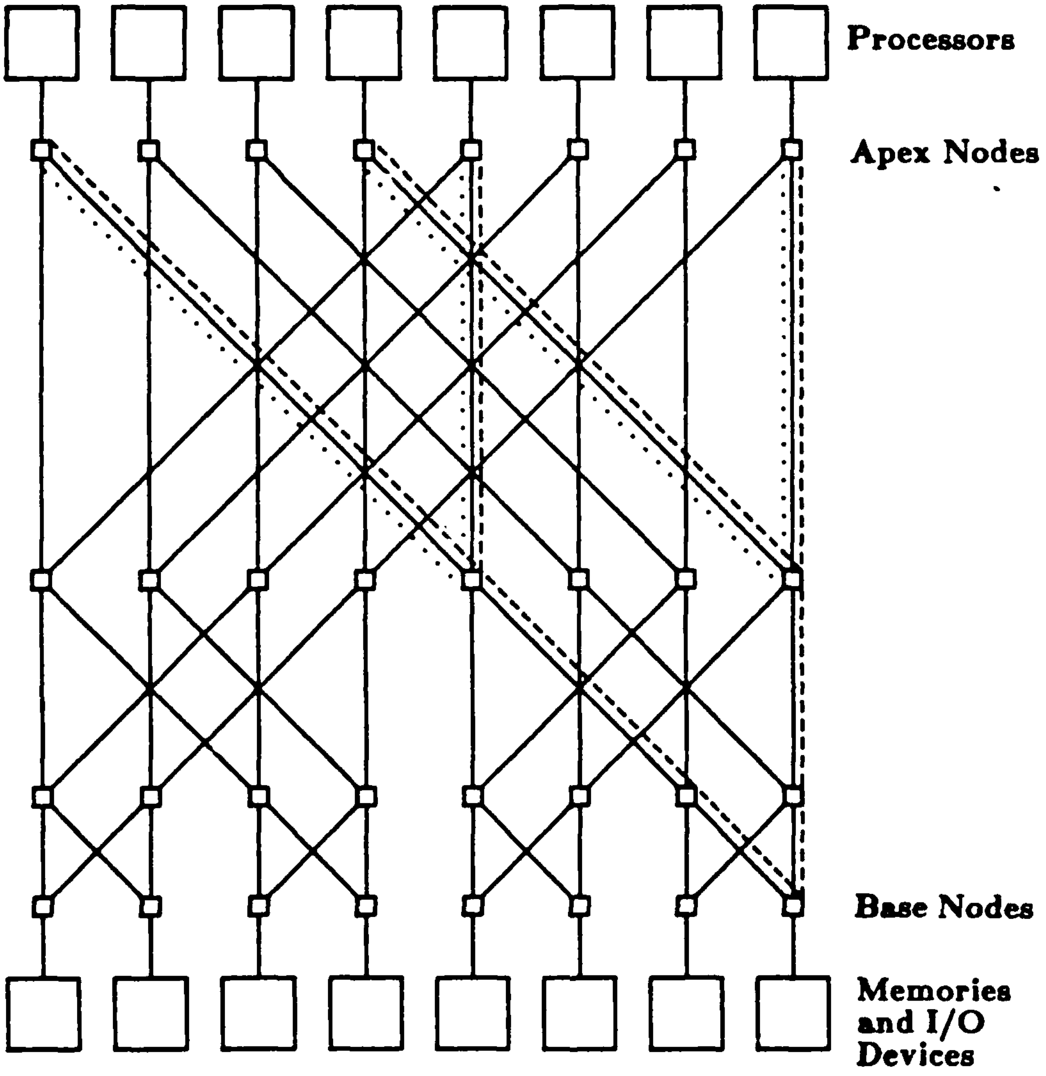


Figure CH.20 Typical banyan interconnection network with spread of 2 and fanout of 2. Solid lines show interconnection links, dashed lines show an instruction tree for a four-processor SIMD machine, and dotted lines show carry-propagation paths.

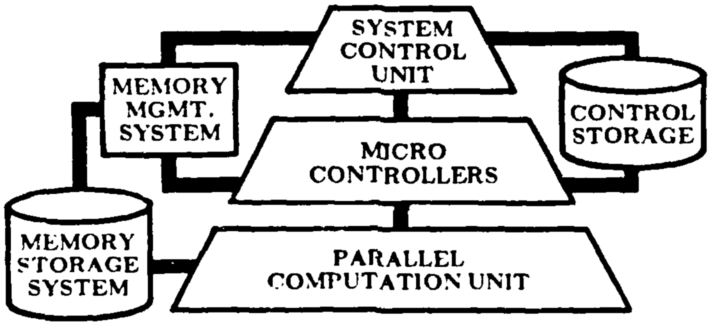


Figure CH.21 Block Diagram of PASM.

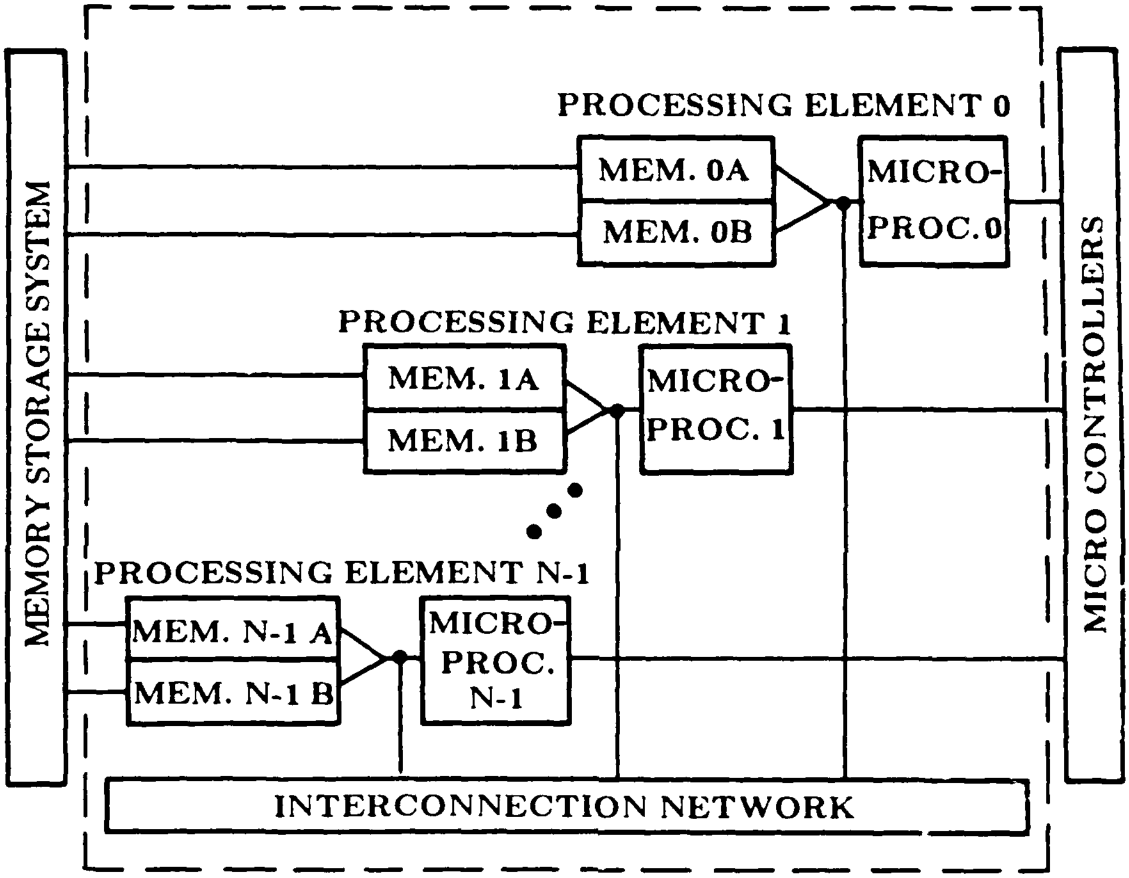


Figure CH.22 Parallel Computation Unit of PASM.

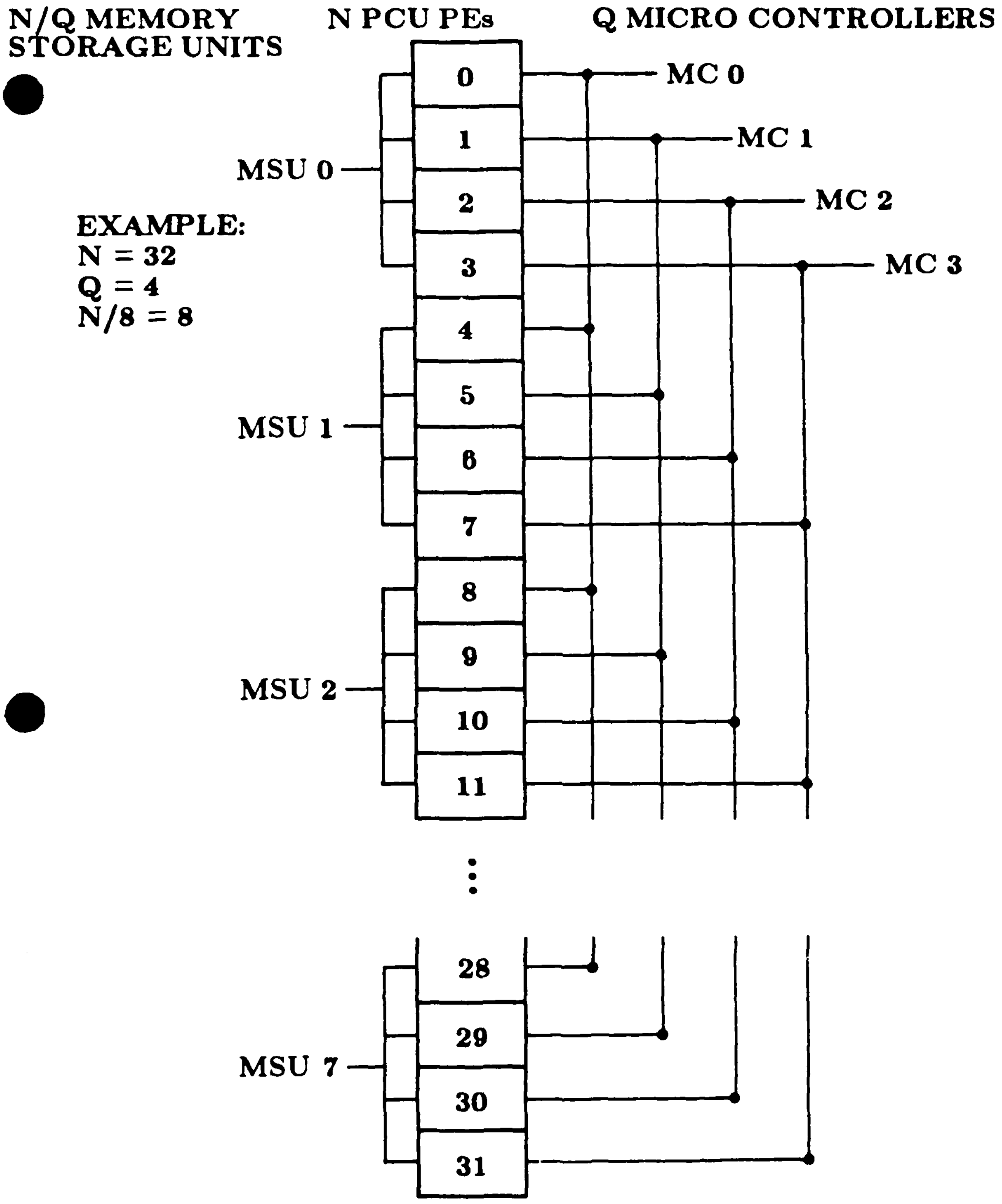


Figure CH.23 Organization of the Memory Storage Unit, shown for N=32 and Q=4. "MSU" is Memory Storage Unit. "PCU" is Parallel Computation Unit.

APPENDIX A.2

ADAPTABLE SOFTWARE FOR SUPERCOMPUTERS

This material was published in

T. Schwederski and H. J. Siegel, "Adaptable software for supercomputers," *Computer,* Vol. 19, February 1986, pp. 40-48.

Adaptable Software for Supercomputers

Thomas Schwederski and Howard Jay Siegel
Purdue University

Software used to control and program reconfigurable supersystems must efficiently exploit the hardware flexibility available. If not, the system does not fulfill its potential.

Steady increases in computer hardware performance show no sign of slowing. Today's supercomputers such as the Cray-1,[1] Cyber 205,[2] and MPP (Massively Parallel Processor)[3,4] are capable of a few million hundred-floating-point operations per second. However, even these computers do not always satisfy the speed requirements of the scientific and defense communities. Consequently, researchers are now exploring new supersystem architectures, such as data flow computers and reconfigurable computers that restructure the organization of their hardware to adapt to computational needs. If a supersystem can be reconfigured, it can more likely efficiently execute tasks that previously required a set of dedicated systems. Reconfiguration is especially important if the programs executed by the supersystem have widely differing computational requirements. Computations needed to control some sophisticated weapon systems are of this variety.[5]

The utilization and performance of any supersystem depends strongly on the software available for it. Necessary software includes system software, such as compilers and operating systems, and application software. Designing the system software to make efficient use of complex supercomputers is difficult. It is further complicated if the supercomputer is reconfigurable. The application programs for such systems are typically very large and complex, such as those for mission-critical military tasks. Thus, two problems facing system designers are

(1) the development of system routines to efficiently control system reconfiguration, and
(2) the writing of complex application programs.

Software used to control and program reconfigurable supersystems must efficiently exploit the hardware flexibility available. If not, then the system does not fulfill its potential. Frequently in the past, customized application software packages were developed for every new computer, often resulting in programs that could be executed on only one type of machine. This machine dependence leads to duplication of effort since the same algorithms have to be coded repeatedly, thus increasing software cost. It is therefore important to look for solutions to these two problems in order to make the efficient use of reconfigurable supersystems practical. We consider possible solutions to these problems.

Adaptable software

Adaptable software offers such a solution. It encompasses all software that manages reconfigurable computer systems and all software itself adaptable to various computers. As noted in the Guest Editors' Introduction, compiler and operating system software used to manage a reconfigurable computer will be termed *reconfiguration software*. Machine-independent software is termed *retargetable software*.

Reconfiguration software is essential for supersystems with reconfigurable ar-

0018-9162/86/0200-0040$01.00 © 1986 IEEE

chitecture. The actual reconfiguration of the hardware must be handled by the operating system of a reconfigurable system to shield the user from details of the hardware implementation and to avoid erroneous hardware settings. The operating system can be directed to perform reconfiguration either explicitly by the user or implicitly by software tools that analyze a given problem automatically, determine the optimum hardware configuration for the problem, and supervise program execution. The explicit approach gives the programmer flexibility in choosing the system configuration needed for a particular algorithm, while the implicit approach reduces programming effort and speeds up program development. Kartashev and Kartashev describe a system that implements the implicit approach for Fortran programs.[6] However, currently no automatic approach exists for all levels of algorithm design and execution. In fact, an exclusively automatic approach may not be desirable for all computer architectures and problems. In some cases explicit reconfiguration enhances efficiency. For example, many highly efficient algorithms for specific computational tasks and specific computer architectures have been developed in the past. The efficiency and speed of such hand-honed algorithms cannot be rivaled by automatically generated programs. For such cases, it is desirable for an adaptable system to assume the architecture for which such an algorithm was designed. To do this, it must provide the user with the tools to easily execute this algorithm on the reconfigurable system.

The life cycle of software can be prolonged if i' is made retargetable. Only programs that do not use machine-specific instructions will be retargetable, since any machine-specific instruction needs to be changed when the program is ported to a different computer. Thus, only programs written in high-level languages will be retargetable.

Similar problems arise for the transportation of programs that use explicit operating system calls. If a program uses explicit operating system calls (which are inherently machine dependent), it is not retargetable. Furthermore, if the operating system is changed or exchanged, the program will have to be rewritten or at least modified to still execute on the same machine. Machine-independent software can survive hardware or operating system changes more easily. The elimination of changes due to hardware or operating system modifications automatically decreases the overall cost of a software system. If a program is portable, it can be used on a wide variety of computers, which eliminates duplicate programming efforts and further reduces cost. Since a machine-independent program is expected to remain useful longer, it is feasible to put more development effort into a program. This results in better, more efficient programs.

We have introduced two different types of adaptable software: retargetable and reconfiguration software. Most reconfiguration software will not be retargetable since it has to be tailored towards machine-specific capabilities. In some cases, components of a reconfiguration software system could be retargetable, such as a reconfiguration control strategy coded in a high level language. Thus, reconfiguration and retargetable software may overlap.

Reconfiguration software

The reconfigurability of a supercomputer can take many different forms. Such a computer could consist of a number of processors with a certain numeric precision. It might be possible to combine several processors to achieve higher precision, such as the reconfigurable varistructure array processor,[7] DCA,[8,9] and TRAC.[10] A supersystem could be switched between the single instruction stream—multiple data stream, or SIMD, and multiple instruction stream—multiple data stream, or MIMD, modes of operation.[11] It could also be partitionable and able to form subgroups of processors. The subgroups could operate in SIMD or MIMD mode, such as PASM.[12] A supercomputer might have even more states, such as array processor, pipeline computer, and multiprocessor, such as DCA[13] and Reddi's and Feustel's machine.[14] The reconfiguration capability might be static, so that the system has to be stopped in order to perform reconfiguration. Alternatively, it could be dynamic, so that architectural states could be switched during program execution. Clearly, static reconfiguration has limited use in mission-critical supersystems since these must perform their tasks at high speed and without interruptions.

The processors of a supersystem must communicate with each other to share data and results. This communication can be handled through shared memory or through processor-to-processor interconnection hardware.[15] In shared-memory systems the processors may be connected to the shared memories through an interconnection network. Examples include C.mmp (which used a crossbar network),[16] CHoPP (which proposed a hypercube structure),[17] and the NYU Ultracomputer (which will use a multistage network).[18] For processor-processor communication a variety of types of networks may also be employed. Hardware interconnections can be fixed as the non-edge connections in the MPP,[3,4] where processors communicate only with their four nearest neighbors. If a processor needs to talk to a processor not its nearest neighbor, multiple data transfers are necessary. Reconfigurable processor-to-processor interconnection networks can allow communication among a large subset or all of the processors, as in PASM,[12] where processors use a multistage cube network. Software running these systems must make efficient use of the interconnection capabilities.

Thus, to take optimal advantage of the system, reconfiguration software needs to know about the hardware's capabilities. It must therefore be tailored towards the hardware on which it will run. If the reconfiguration software is retargetable, the machine-specific hardware description would have to be entered at compile or run time, since it could not be part of the retargetable software.

Reconfiguration of a supersystem can be handled by the programmer through explicit use of reconfiguring instructions, or implicitly by system software which automatically generates reconfiguring instructions without the need of programmer intervention. Hybrids of these methods are also possible.

Explicit reconfiguration. If instructions for reconfiguring the system are available to the user, the programmer can take explicit advantage of various architectural states. With careful program analysis and good coding, this results in programs gaining the most advantage from the reconfiguration capability. Programs for each

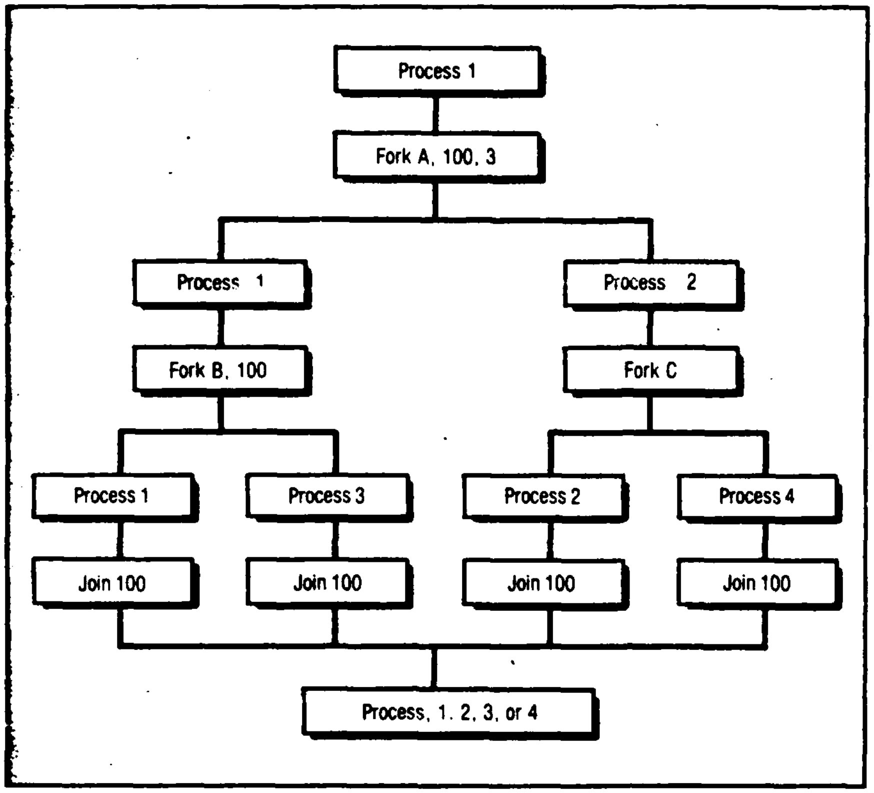


Figure 1. Flowchart for a parallel program using **Fork** and **Join** statements.

architectural state like array, multiprocessor, pipeline, etc., must be written in a language suited for this state, and the language must include constructs for reconfiguration. High level languages for MIMD and SIMD machines give good examples of the specific requirements. Consider an MIMD computer. If the number of processors for the given task is fixed, a standard assembly or high level language such as Pascal, C, or Fortran can be used to program the task. The programmer has to decide which processor handles what particular task, and write the appropriate programs. The programs are loaded into the processors needed for the task. The operating system of the machine starts the task and coordinates activities. Many applications, especially in image processing, require a set of processors that all execute the same program. An example is an image processing algorithm in which each processor finds features in a part of the image. In this case, each processor has its own set of data, but all processors execute the same program. Only one program has to be coded, independent of the number of processors. Depending on the task, this may involve just MIMD operations, just SIMD operations, or both.[19]

If the number of processors varies during task execution, the programming language has to be expanded and must include special statements to allocate and deallocate processors. Conway[20] proposes the use of **Fork** and **Join** instructions. Whenever a **Fork** is encountered, a new process is created that proceeds independently from the original process. The new process can run on another processor. If no processor is available, the new process can be queued until a processor becomes available. If multiprogramming capability is available, such as CHoPP,[17] two or more processes can run on the same processor in a timesliced fashion. Conway proposes three kinds of **Fork: Fork A, J, N**; **Fork A, J**; and **Fork A**. In this implementation, a counter keeps track of the number of concurrent processes. **Fork A** generates a new process; the old process continues to execute the instruction following the **Fork**, the new process starts at instruction A. The **Fork A, J, N** instruction sets a counter in memory location J to the value N, then executes **Fork A**. The programmer must match the number of **Forks** in his program with the count specified in the **Fork A, J, N**, since the **Fork A** alone does not affect the counter. **Fork A, J** increments counter J by one and then executes a **Fork A**. This instruction is needed if the decision to fork is made at run time. The **Join** statement merges concurrent instruction streams. When a **Join J** is encountered, the counter at location J is decremented. The processor that decrements the counter to zero executes the statements following the **Join**. All other processors are released and become available for other computations.

Figure 1 illustrates the use of the three different **Fork** and the **Join** statements. Process 1 executes a **Fork A, 100, 3**, which sets the counter in memory location 100 to 3 and starts a concurrent process (Process 2). Then Process 1 executes a **Fork B, 100**, which increments the counter at location 100 by one and starts another process, Process 3. Meanwhile, Process 2 executes a **Fork C**, generating a fourth process, Process 4. Thus, four concurrent processes are now active. When a process encounters a **Join** 100, it decrements the counter. The process continues if the counter reaches zero; otherwise, it ceases. **Fork** and **Join** are examples of two explicit reconfiguration commands.

Dijkstra[21] proposed a more structured way to express reconfiguration for multiprocessor programs: **parbegin** *S1;S2;*.... *Sn*; defines *S1* through *Sn* as statements which can be executed in parallel. The parallel execution ends when a **parend** is found. Only after all processors executing the parallel statements have found their **parend** are statements following the parend executed. A program that needs to run two processes in parallel might have the following structure:

```
parbegin
process1 : begin
  statements for process 1
  end
process2 : begin
  statements for process 2
  end
parend
```

Both **parbegin-parend** and **Fork-Join** blocks could be conditional. In other words, whether or not the block would be entered could depend on the result of an **if** statement. This results in a dynamic reconfiguration capability.

Parallelism of an SIMD computer cannot be expressed explicitly in any conventional programming language. One type of SIMD computer consists of an array of processor/memor,' pairs called processing elements, or PEs, which do the actual computations, and a control unit. The control unit executes all program flow instructions and broadcasts data processing instructions to the PEs. All PEs execute the same instruction at the same time, but on different data, e.g., on different sections of an image. The control unit also determines which of the PEs execute the instruction and thereby varies the extent of parallelism of the computer. An interconnection network provides communication between the PEs. The network can be fixed as in the MPP or the Illiac IV,[22] or it can be reconfigurable as in PASM.[12] If the network is reconfigurable, instructions that reconfigure the network in the desired fashion must be provided in the programming language. Constructs to enable and disable PEs need to be available as well. Extensions of existing high level languages have been proposed.[23-30] Such extensions can be machine dependent or machine independent.

An example of a machine-dependent language is Glypnir,[24] specifically developed for the Illiac IV computer and based on Algol 60. Illiac IV was an SIMD machine with 64 PEs, designed in the 1960's. Its fixed interconnection network connects PE i to PEs i + 1, i − 1, i + 8, i − 8, all modulo 64, providing a mesh-type interconnection. The system cannot be partitioned into smaller machines; the only reconfiguration capability involves enabling and disabling PEs. In Glypnir, variables for PEs and for the control unit can be defined. A PE variable has an element in every PE and is called a super word, or *sword*. Therefore, a sword always has 64 elements. For example, the statement

```
PE REAL A,B
```

declares the variables *A* and *B* to be swords.

```
PE REAL C(20)
```

declares *C* to be an array of 20 elements in each of the PEs. By using Boolean expressions, PEs can be selectively enabled or disabled. Consider, for example, the statement

```
if <Boolean expression> then
   <statement1>
else
   <statement2>
```

The Boolean expression will be evaluated in all processors. The processors finding a true value execute statement1. The others do not participate in the instruction stream broadcast for statement1, but execute only statement2.

Glypnir's architectural dependence on the Illiac IV severely limits its use. An attempt for machine independence was made by the developers of Actus,[26] a language based on Pascal. Actus was designed for SIMD machines of arbitrary size. A variable can be declared parallel. The extent of parallelism must be specified for each parallel variable. In the range specification of a parallel variable declaration, a colon denotes that a dimension is to be accessed in parallel. For example,

```
var array1: array [1:m,1..n] of integer
```

declares an array of *m* by *n* elements; *m* elements can be accessed at a time. The underlying operating system takes care of reconfiguring the computer into a machine that can handle the specified parallel variable. **if**, **case**, **while**, and **for** statements are expanded for use with parallel variables. Alignment operators are also provided. For example, a **shift** operator moves data within the declared range of parallelism, a **rotate** shifts data circularly with respect to the extent of parallelism. **shift** and **rotate** will be implemented by using the processor interconnection network.

Parallel-C is a proposed programming language based on C that handles both the MIMD and SIMD modes of operation.[30] The SIMD features include declarations of parallel variables and functions; an indexing scheme to access and manipulate parallel variables; expressions involving parallel variables; extended control structures using parallel variables as control variables; and functions for PE allocation, data alignment, and I/O. The language supports simple parallel data types like scalars and arrays as well as structured data. For example,

```
parallel [N] int a;
parallel [N] char line[MAXLINE];
struct node {
   char *word; struct node *next;
} parallel [N] nodespace[100], *head;
```

declares an integer "a," an array of MAXLINE characters, an array of 100 nodes, and a node pointer for each of the N processors. Indexing along the parallel dimension can be done by using selectors; they enable or disable subsets of the N processors. Functions to send data between processors are provided. They are not part of the language, but library routines, thus avoiding machine dependence. If the compiler is retargeted, these communication routines must be adapted. No change of the compiler itself is required. To facilitate MIMD mode, a few new features and keywords are added. A preprocessor recognizes these constructs and translates the parallel algorithm into a standard serial C program, which can then be compiled by a standard C compiler and loaded into the program memories of the processors of the MIMD machine.

Implicit reconfiguration. On a supercomputer with a fixed architecture, it is desirable to have the ability to automatically analyze a serial program and restructure it so that it can be executed efficiently. As reconfigurable supercomputers become more and more complex, having users use explicit reconfiguration commands to construct efficient algorithms for the system may no longer be feasible. On a machine with reconfigurable architecture, it is desirable to automate the process of finding an optimal architectural state and then structuring the task for the architecture. This may involve more than one architec-

A task may be most efficiently performed using different architectural states for different subtasks.

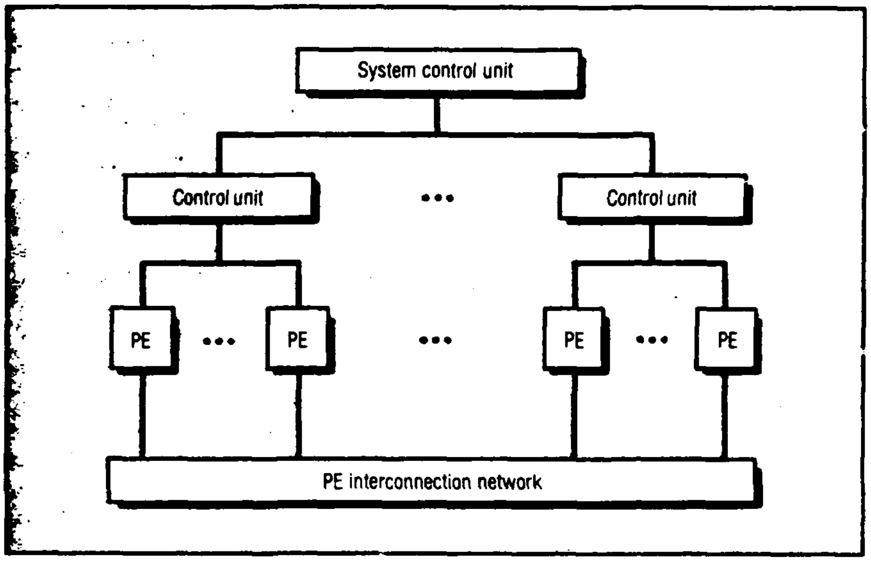


Figure 2 Simplified block diagram of PASM.

tural state, because a task may be most efficiently performed using different architectural states for different subtasks.

Much effort has gone into developing algorithms capable of detecting parallelism in serial programs.[31,32] Consider a program to add to vectors A and B:

FOR i = 1 **TO** N **DO** C[i] := A[i] + B[i];

Obviously, the addition statements do not depend on each other; therefore, all additions can be executed in parallel on N processors. This example typifies the operations performed by a parallelizing compiler. The compiler analyzes the program and schedules statements that do not depend on each other to exeecute in parallel. It must also determine the number of processors needed to execute the program and have the operating system allocate them. Since the order in which two statements, I and J, execute in a sequential program is not preserved if they are executed in parallel, three conditions must be met if I and J are to be parallelized. Statement I must not write to locations that statement J reads; statement J must not write to locations that statement I reads; and statement I must not write to the same location as statement J if that location is used by a later statement.

One approach to handle a system with reconfigurable architecture, developed by Kartashev,[6] is called the reconfiguration flowchart. The algorithm assumes a dynamic architecture that can partition its processors into groups of computers of different sizes and can combine processors to form higher-precision computers. It analyzes a Fortran program and its program flow structure and constructs a program graph. Each node in the graph includes one or more program statements. The arrows between nodes represent the program flow. For each node, the necessary computational precision is determined and by that the minimum number of processors required for the computation of the node is found. Since a minimum number of processors is found for each node, a maximum number of nodes can be processed simultaneously.

Several programs can be run concurrently on the system. Whenever a program finishes, a new program should be started, necessitating a switch to a new architectural state. If the task causing the switch runs for a longer time than other tasks, the processors executing the short tasks will wait for the long task to finish. Therefore, the execution time for each node is determined to minimize processor idle time. Then a resource diagram is constructed, where nodes are assigned specific processors, and execution can begin.

Ongoing research at Purdue involves the design of expert-system-based intelligent operating systems for controlling reconfigurable supersystems.[33] To optimize system performance, the intelligent operating system uses information about which subtasks need to be executed to perform an overall task, the performance/system-requirement characteristics of the algorithms for the subtasks, and the current state of the system.

Reconfiguration methodology. An important consideration in reconfigurable systems is the reconfiguration methodology. The switch from one architectural state to another takes time. To maximize performance, the sum of execution time and reconfiguration time must be minimized. Clearly, a very short reconfiguration time is desirable, since otherwise advantages gained by the reconfiguration are lost due to the reconfiguration overhead. Of course, the time to reconfigure has to be less than (hopefully, much less than) the reduction in execution time gained through the reconfiguration.

As an example of a reconfiguration methodology, consider the PASM system. PASM is a partitionable SIMD/MIMD system. A prototype with 16 Motorola MC68000-based PEs in the computational engine and four control units is being constructed at Purdue University.[34] Figure 2 shows a simplified block diagram of the system. In SIMD mode, the PEs receive their instructions from a control unit and process data from their private memory. In MIMD mode, the PEs have data and program in their private memory. One type of reconfiguration is the switching from SIMD to MIMD mode and vice versa. Whenever a PE addresses a certain address range, AR, this PE is in SIMD mode. As soon as an access to AR is detected, an instruction request is issued to the control unit, which then broadcasts an instruction to the requesting processors. The control unit accomplishes a switch from SIMD to MIMD mode by broadcasting an unconditional jump to the beginning of the MIMD program to the PEs, where the address of this MIMD program is outside the range AR. The PEs can return to SIMD mode by jumping into the address space AR. The reconfiguration overhead for PASM is therefore just one instruction. For all practical applications, this overhead is negligible and program sections can be executed in SIMD or MIMD mode, whichever is best suited.

Apart from reconfiguration for optimum performance, reconfiguration due to faults or other undesired conditions such as unbalanced load must be considered in a supersystem. If a single processing element of a supersystem fails, the system as a whole should continue to function, eventually with decreased performance. The necessary reconfiguration has to be automatic and must be handled by the operating system, since a programmer cannot predict a fault. Ma[35] proposes reconfiguration control algorithms for reconfigurable computers with and without centralized control. As an example for the methodology, consider the straightforward reconfiguration procedure with centralized control. The control unit monitors the state of the processing nodes and detects undesirable conditions. It then broadcasts reconfiguration commands to the appropriate nodes, which perform the necessary reconfiguration and signal completion to the control unit. After all nodes have completed their respective reconfiguration tasks, the control unit signals the nodes and computation can resume.

An additional consideration for reconfiguration software is the ability to concentrate computing power.[33] For example, assume that numerous subtasks of some overall task are being executed concurrently. If, as a result of some derived intermediate result, one subtask becomes more time-critical than the others, the system should dynamically reconfigure to provide additional resources to the subtask. Reconfiguration software capable of such operation would have to have some form of intelligence and knowledge of a system model. This is an area for future research.

Retargetable software

As mentioned previously, software that can be used on a wide variety of computers is very desirable. An early attempt towards that goal is the standardization of the programming language Fortran by ANSI. Nevertheless, many Fortran dialects exist, and compilers are not verified. Therefore, it cannot be guaranteed that the same program will exhibit identical performance on different machines. A standard Fortran program using only standard I/O routines like **read** and **write** might very well be portable. As soon as explicit operating system calls such as **seek** for a disk are included, however, the program is restricted to machines with compatible operating system calls.

Other important tasks that depend on the operating system are task creation and interprocess communication. They are necessary for expressing concurrent operations. Concurrent operations are often used in real-time systems and are essential for parallel processing. Low-level operations like interrupt control are not possible in Fortran. All this seriously limits the portability of Fortran programs. The same problems arise with other languages, such as Pascal and Algol. Current efforts on the design of Ada[36,37] are attempting to circumvent these problems.

There are basically three ways to make a program portable.[38]

Transportable language approach. This approach, designed for serial computers, is based on widely used languages like Fortran and Pascal. The dialects of such a language are analyzed and a hypothetical parent language with a grammar common to all dialects is defined. The syntax and semantics of the parent language are rigorously specified. For each computer which is to run portable code, parameters which completely specify the computer architecture are determined. These parameters are byte size, word size, byte or word orientation of the CPU, main memory size, and maximum program module size. Then a compiler with two modes of operation is constructed: the transportability testing mode and the compilation mode. In the transportability testing mode, a program coded in a dialect of the language is converted to the hypothetical parent language, using the architectural parameters. In the compilation mode, hypothetical parent-language code is translated into a dialect of the language. Thus, existing programs can be converted to the parent language, making them portable. Also, programs can be written directly in the parent language and thus be guaranteed easily portable.

Certain instructions must be modified for the parent language. For example, all variable declarations need to include a precision declaration so that the parent language compiler can determine the appropriate data type in the language dialect. Desirable language features not available in a given dialect (such as data structures or **while** loops) can be made available in the parent language, thus making programming in the parent language more flexible.

A major advantage of the transportable language approach is that existing programs are not rendered obsolete.

A major advantage of the transportable language approach is that existing programs are not rendered obsolete but can be converted and made portable. The necessity to write bifunctional compilers for every machine, and the continued existence of multiple dialects for the language are disadvantages.

Emulation approach. Another approach to transportable software is to transform the machine rather than the program. Making machines capable of emulating some class of computers allows those machines to execute code written for any of the computers in that class.[39,40] Programs can be coded in any high-level or assembly language supported by the computers in that class. The programs are compiled and the resulting machine code is transferred to one of the emulation machines. This machine emulates the computer for which the code was written and thus is capable of executing the machine code. One problem with the approach is decreased performance. Another is that all machines, of a set of machines among which code is to be transportable, have to be capable of emulating all the others.

New language approach. In this approach, all programs to be ported are written in a standard new language known by all machines, thus making programs portable. The definition of this new language incorporates all desired features and the compilers for this language are rigorously verified to ensure portability. The Department of Defense chose this method when

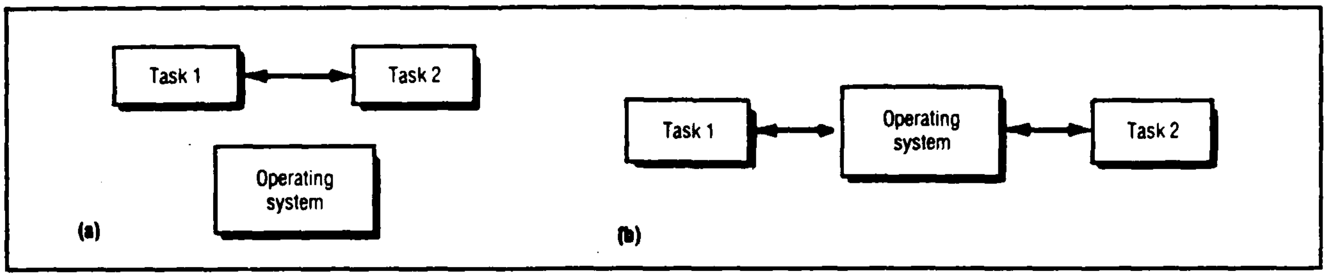


Figure 3. (a) User's view of task communication in Ada. (b) User's view of task communication in a standard programming language.

developing the programming language Ada.[37,41,42]

Ada is a structured programming language. In addition to being designed for portability, it supports many features necessary for executing tasks on reconfigurable systems. Functions necessary for process creation, deletion, and interprocess communication are part of the language. The compiler actually generates operating system calls from those functions, but this is hidden from the user. If the program is transferred to a different machine and recompiled, the appropriate new operating system calls are generated.

Depending on the machine architecture, concurrent processes could either be run in a time-sliced mode, or they could be allocated on different processors. The compiler for a particular machine, in cooperation with the operating system, has to handle the allocation procedure.

Concurrent processes are called tasks in Ada. A task is declared by defining a task header and a task body. If one or more tasks are declared inside a procedure, they execute concurrently with the procedure. The task header specifies the task name and entries that handle interprocess communication. The task body contains all statements executed in the task. To send a message to task T, other tasks call an entry of T like a procedure. The destination task T can accept the message.

As an example, consider a process that accepts single-character messages from other tasks. The header has the form

```
task ACCEPT_CHAR is
   entry MESSAGE ( C: in
   CHARACTER)
end;
```

The task name is defined as ACCEPT_CHAR. Other processes can send messages to ACCEPT_CHAR calling the entry:

```
MESSAGE (CHAR);
```

The task ACCEPT_CHAR reads the message when it encounters an **accept** statement, which must be part of the task body:

```
accept MESSAGE (C: in
   CHARACTER) do
   MESS := C
   end;
```

This statement accepts an incoming message and assigns its value to the variable MESS. Only between the **do-end** part of the **accept** is the passed message accessible. Data is actually exchanged only when the sending task calls an entry and the receiving task executes an **accept** on the same entry. If a process encounters an **accept** without another process executing the appropriate entry call, the process will wait until an entry call is executed. An entry call also waits for the appropriate **accept**. Thus, sending and accepting of messages is synchronized between tasks.

The execution of an entry call and its associated **accept** statement is called *task rendezvous*. The task rendezvous facilitates interprocess communication by the ability to exchange data. It facilitates process synchronization by executing the task only if both the sending and receiving tasks have reached their respective rendezvous statements. It facilitates mutual exclusion since only one of all tasks requesting communication with another process will be served at any given time. The difference between an Ada task rendezvous and task communication in other programming languages is illustrated in Figure 3. Since no explicit operating system calls are needed, the task handling is not machine specific and can easily be ported.

Another important portability problem with programming languages arises from machine-dependent data types. A Cray-1 supercomputer,[2] for example, has an integer precision of 24 bits, whereas the Fujitsu VP-200[43] uses 32-bit integers. Programs utilizing 32-bit integer precision could not run on a machine of 16-bit integer precision. In Ada, a programmer has access to such machine-dependent datatype attributes as

```
INTEGER_SIZE
```

Using such attributes, a program can check at runtime whether a specific machine will be able to execute a program as desired.

The task concept of Ada can be used to explicitly handle parallel processing in MIMD multiprocessors. This is insufficient to express parallelism of an SIMD machine. Extensions to Ada facilitating SIMD programming have been proposed[29]; these extensions provide functions similar to those for SIMD programming as described above. Due to the rigid Ada policy allowing no subsets or supersets of the language in a verified compiler, this approach may not be widely used. To code SIMD programs explicitly nevertheless, subprograms coded in a language suitable for SIMD programming, such as Actus or extended Ada, could replace a standard Ada subprogram if the program is executed on an SIMD machine. This procedure is analogous to replacing a high-level language subroutine by a faster executing assembly program. The main body of the program would then still be portable. Only the special subroutine has to be changed when the program is retargeted.

As more and more adaptable supersystems are designed and built, the need for adaptable software to run these systems efficiently will continue to grow. The adaptable software required includes retargetable software, which makes application programs transportable, and reconfiguration software, which controls the system organization. Thus, adaptable software is an essential element in the development of supersystems or mission-critical objectives. □

Acknowledgments

The authors would like to thank Steven and Svetlana Kartashev and Andre van Tilborg for their careful reading of the manuscript and many helpful suggestions.

This work was supported by the Rome Air Development Center under contract number F30602-83-K-0119.

References

1. R. M. Russell, "The Cray-1 Computer System," *Comm. ACM*, Jan. 1978, pp. 63-72.
2. E. W. Kozdrowicki and D. J. Theis, "Second Generation of Vector Supercomputers," *Computer*, Nov. 1980, pp. 71-83.
3. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers*, Vol. C-29, Sept. 1980, pp. 836-844.
4. K. E. Batcher, "Bit Serial Parallel Processing Systems," *IEEE Trans. Computers*, Vol. C-31, May 1982, pp. 377-384.
5. E. W. Martin, "Strategy for a DoD Software Initiative," *Computer*, Vol. 16, Mar. 1983, pp. 52-59.
6. S. P. Kartashev and S. I. Kartashev, "Distribution of Programs for a System with Dynamic Architecture," *IEEE Trans. Computers*, Vol. C-31, June 1982, pp. 488-514.
7. G. J. Lipovski and A. R. Tripathi, "A Reconfigurable Varistructure Array Processor," *1977 Intl. Conf. Parallel Processing*, Aug. 1977, pp. 165-174.
8. S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," *Computer*, July 1978, pp. 26-42.
9. S. I. Kartashev and S. P. Kartashev, "A Multicomputer System with Dynamic Architecture," *IEEE Trans. Computers*, Vol. C-28, Oct. 1979, pp. 704-720.
10. G. J. Lipovski, "The Banyan Switch in TRAC the Texas Reconfigurable Array Computer," *Distributed Processing Technical Committee Newsletter* (*IEEE Computer Society*), Jan. 1984, pp. 13- 26.
11. J. Keng and K-S. Fu, "A Special Computer Architecture for Image Processing," *1978 IEEE Computer Society Conf. Pattern Recognition and Image Processing*, June 1978, pp. 287-290.
12. H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Computers*, Vol. C-30, Dec. 1981, pp. 934-947.
13. S. I. Kartashev and S. P. Kartashev, "Problems of Designing Supersystems with Dynamic Architecture," *IEEE Trans. Computers*, Vol. C-29, Dec. 1980, pp. 1114-1132.
14. S. S. Reddi and E. A. Feustel, "A Restructurable Computer System," *IEEE Trans. Computers*, Jan. 1978, pp. 1-20.
15. H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.
16. W. Wulf and C. Bell, "C.mmp—A Multi-Miniprocessor," *AFIPS 1972 Fall Joint Computer Conf.*, Dec. 1972, pp. 765-777.
17. H. Sullivan, T. R. Bashkow, and K. Klappholz, "A Large-Scale Homogeneous, Fully Distributed Parallel Machine," *Fourth Ann. Symp. Computer Architecture*, Mar. 1977, pp. 105-124.
18. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared-Memory Parallel Computer," *IEEE Trans. Computers*, Vol. C-32, Feb. 1983, pp. 175-189.
19. D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A Parallel Algorithm for Contour Extraction: Advantages and Architectural Implications," *1983 IEEE Computer Society Symp. Computer Vision and Pattern Recognition*, June 1983, pp. 336-344.
20. M. E. Conway, "A Multiprocessor System Design," *AFIPS 1963 Fall Joint Computer Conf.*, 1963, pp. 139-146.
21. E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, ed. F. Genuys, Academic Press, New York, NY, 1968, pp. 43-112.
22. W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV System," *Proc. IEEE*, Vol. 60, Apr. 1972, pp. 369-388.
23. K. G. Stevens, "CFD—A Fortran-like language for the Illiac IV," *ACM Conf. Programming Languages and Compilers for Parallel and Vector Machines*, Mar. 1975, pp. 72-76.
24. D. H. Lawrie, T. Layman, D. Baer, and J. M. Randall, "Glypnir—A Programming Language for Illiac IV," *Comm. ACM*, Vol. 18, Mar. 1975, pp. 157-164.
25. S. F. Reddaway, "DAP—A Distributed Array Processor," *1st Ann. Symp. Computer Architecture*, Dec. 1973, pp. 61-65.
26. R. H. Perrott, "A Language for Array and Vector Processors," *ACM Trans. Programming Languages and Systems*, Vol. 1, Oct. 1979, pp. 177-195.
27. L. Uhr, "A Language for Parallel Processing of Arrays, Embedded in Pascal," *Languages and Architectures for Image Processing*, eds. M. J. B. Duff and S. Levialdi, Academic Press, London, England, 1981, pp. 53-88.
28. A. P. Reeves and J. D. Bruner, "The Programming Language Parallel Pascal," *1980 Int'l Conf. Parallel Processing*, Aug. 1980, pp. 5-7.
29. C. Cline and H. J. Siegel, "Augmenting Ada for SIMD Parallel Processing," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 9, Sept. 1985, pp. 970-977.
30. J. T. Kuehn and H. J. Siegel, "Extensions to the C Programming Language for SIMD/MIMD Parallelism," *1985 Int'l Conf. Parallel Processing*, Aug. 1985, pp. 232-235.

31. D. J. Kuck and D. A. Padua, "High-Speed Multiprocessors and Their Compilers," *1979 Int'l Conf. Parallel Processing*, Aug. 1979, pp. 5-16.

32. Arvind, "Decomposing a Program for Multiple Processor Systems," *1979 Int'l Conf. Parallel Processing*, Aug. 1979, pp. 7-14.

33. E. J. Delp, H. J. Siegel, A. Whinston, and L. H. Jamieson, "An Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement*, Nov. 1985, pp. 217-224.

34. H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: A Reconfigurable Parallel System for Image Processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition*, July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter, *Computer Architecture News*, Vol. 12, No. 4, Sept. 1984, pp. 7-19.)

35. Y. W. Ma, "Reconfiguration Control Algorithms for Reconfigurable Computer Systems," *Int'l Computer Software and Applications Conf.*, Nov. 1982, pp. 70-77.

36. J. G. P. Barnes, *Programming in Ada*, Addison-Wesley, London, England, 1982.

37. U.S. Department of Defense, *Reference Manual for the Ada Programming Language*. Washington, D.C., 1982.

38. P. A. D. de Maine and S. Leong. "Transportation of Programs," *15th Southeast Symp. System Theory*, Mar. 1983, pp. 158-164.

39. T. A. Marsland and J. C. Demco, "A Case Study of Computer Emulation," *Canadian J. Operational Research and Information Processing*, Vol. 16, June 1978, pp. 112-131.

40. J. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, NY, 1978.

41. H. Ledgard, *Ada, An Introduction*, Springer-Verlag, New York, NY, 1983.

42. R. J. A. Buhr, *System Design with Ada*, Prentice Hall, Englewood City, NJ, 1984.

43. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.

Thomas Schwederski is currently working toward a PhD degree at Purdue University. He received the Diplom-Ingenieur degree in electrical engineering from Ruhr-Universitaet Bochum in 1983, and the MS degree in electrical engineering from Purdue University in 1985. His research interests include computer architecture, parallel processing, multimicroprocessing systems, and VLSI design.

Howard Jay Siegel is a professor of electrical engineering and Director of the PASM Parallel Processing Laboratory at Purdue University, where he has been involved in research and teaching since 1976. His research interests include parallel/distributed processing, computer architecture, and image and speech understanding.

Siegel received BS degrees in electrical engineering and management from the Massachusetts Institute of Technology in 1972. He received the MA and MSE degrees in 1974, and the PhD degree in 1977, all three in electrical engineering and computer science, from Princeton University.

Siegel has served as an IEEE Computer Society distinguished visitor and is currently an associate editor of the *Journal of Parallel and Distributed Computing*.

Questions regarding this article can be directed to either author at the PASM Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

APPENDIX B

DESIGN OF A 1024-PROCESSOR PASM SYSTEM

This material was published in

J. T. Kuehn, T. Schwederski, and H. J. Siegel, "Design of a 1024-processor PASM system," *First International Conference on Supercomputing Systems,* December 1985, pp. 603-612.

From the Proceedings of the First International Conference on Supercomputing Systems, December 1985.

DESIGN OF A 1024-PROCESSOR PASM SYSTEM*

James T. Kuehn
Thomas Schwederski
Howard Jay Siegel

PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907 USA

Abstract

PASM is a multifunction SIMD/MIMD computer system being developed at Purdue University for image and speech understanding. A 30-processor prototype (with 16 processors in the computational engine) based on Motorola 68000-family components is currently under construction. This paper proposes a design for a "full" PASM system consisting of 1024 processors using the lessons learned from the PASM prototype as a guide. Specific performance goals are outlined and the hardware required to achieve them are described.

I. Introduction

Among the variety of supercomputer architectures that have been proposed or constructed, there is one common characteristic: the use of parallelism to increase performance. Scientific attached processors such as the AP-120B [1] use pipelining and sophisticated floating point multiply-add units to perform vector operations and have performance of 10-50 million operations per second (MOPS)**. Vector supercomputers such as the Cray-1 [2] and Cyber 205 [3] have multiple vector pipelines and achieve processing rates of a few hundred MOPS. Enhancements to these vector supercomputers employing multiprocessing concepts are being developed. Arrays of bit-serial processors such as STARAN [4] and MPP [5] operate in single instruction stream - multiple data stream (SIMD) mode [6]. MPP may achieve performance exceeding 1000 MOPS for highly structured tasks such as image processing. Multiprocessor systems such as HEP [7] and S-1 [8] have complex processors and operate in multiple instruction stream - multiple data stream (MIMD) mode [6]. The performance of multiprocessors varies widely depending on the number and type of CPUs, interconnection scheme, and operating system. However, most existing multiprocessors have no more than 16 CPUs and do not yet achieve the performance of the vector supercomputers. Our goal at Purdue is the design of PASM, a partitionable SIMD/MIMD processing system that can be dynamically reconfigured to meet the particular processing needs of a large variety of applications in the image and speech analysis domains.

SIMD parallelism has shown tremendous performance for structured tasks with large data sets. The early stages of image processing tasks that consider the pixel-level representation of an image such as clipping, smoothing, histogramming, and 2-D correlation, are examples of highly efficient SIMD algorithms. MIMD parallelism is more suited for the later stages of image processing tasks that deal with higher-level image representations such as lines, contours, and regions. PASM is being designed to work in both modes of parallelism so that the same processors can be used to perform complete tasks. Also, each of the algorithms which comprise the task may be coded to utilize the most efficient mode of parallelism

An *SIMD machine* typically consists of a *Control Unit (CU)*, an interconnection network, and $N=2^n$ *Processing Elements (PEs)*, where each PE is a processor/memory pair. The PEs are numbered from 0 to N-1 and each PE knows its number (address). This is shown in Figure 1. The CU broadcasts instructions to the processors and *all* active (enabled) processors execute the same instruction at the same time. This is the single instruction stream. Each processor operates on data taken from a memory to which only it is connected. These represent the multiple data streams. The interconnection network allows interprocessor communication. Examples of such machines are the Illiac IV [9], STARAN [4], CLIP4 [10], and MPP [11]. An *MIMD machine* has a similar organization, but each processor may *follow an independent instruction stream*. As with SIMD architectures, there is a multiple data stream and an interconnection network. Examples of such machines are C.mmp [12], Cm* [13], Ultracomputer [14], and HEP [7].

An *MSIMD (multiple-SIMD) system* is a parallel processing system which can be structured as one or more independent SIMD machines (e.g., MAP [15]). A *partitionable SIMD/MIMD system* is a parallel processing system which can be structured as one or more independent SIMD and/or MIMD machines (e.g., TRAC [16], PASM [17, 18]) PASM is being developed using a variety of applications problems from the areas of image and speech analysis to

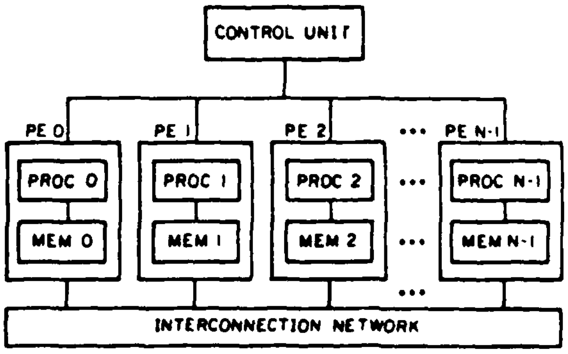


Figure 1. SIMD/MIMD machine model.

*This research was supported by the Rome Air Development Center under contract number F30602-83-K-0119 and by an IBM Graduate Fellowship.

** For this paper, an "operation" (OP) will be defined to consist of fetching a 32-bit (or comparable size) integer value from memory using an address in an address register, adding the value to a 32-bit integer data register value, and leaving the result in a data register. For pipelined and vector machines, steady-state performance without memory contention is assumed. A "floating operation" (FLOP) is similar but assumes 64-bit floating point values.

guide the machine design choices. A 30-processor prototype PASM system (including 16 PEs) based on the Motorola MC68010 microprocessor and other "off-the-shelf" components has been designed and is currently under construction [19]. The prototype is a vehicle for application studies, parallel high-level language development, and distributed operating systems research. It will achieve a performance of about 10 MOPS. This paper draws on the experience of the PASM prototype and proposes a design and implementation for a 1024-PE PASM system.

This research is motivated by the desire to validate the scalability of PASM. Three different forms of scalability can be identified. *Architecture scalability* requires that the potential computing power increase linearly with N, the number of processors, and cost and number of interconnections increase as N log N at worst, and preferably linearly. For example, a crossbar network in which the number of connection points grows as N^2 does not have architectural scalability. A bus shared by all the processors in a system is not architecturally scalable because its fixed bandwidth prevents the potential computing power from increasing linearly with N. The hierarchical PASM organization with its $O(N\log_2N)$-connection multistage network achieves this architecture scalability.

Algorithm scalability is implied if increasing the number of processors used to perform an algorithm results in an improved performance (using execution time, throughput, or some other measure). Increasing N beyond the problem size *P* (expressed as the number of fundamental operations) clearly does not result in a performance increase; therefore, algorithm scalability is defined so long as $N \leq P$. Our earlier algorithm complexity and simulation studies have shown that parallel image processing algorithms could effectively use hundreds or thousands of processors.

Implementation scalability in the narrow sense means that components can be added to a system without making changes to the existing implementation scheme. This is rather difficult to achieve in a design because cost and expediency often make a small system radically different from a larger one. An example of narrow sense scalability is a distributed computer system in which processors communicate only with a few of their immediate neighbors: processors can be added indefinitely to such a system. Implementation scalability in the wide sense means that the nature and function of the basic components does not change as the machine size changes, but the technology employed may change. For example, PEs in a small system may each be implemented with several physical boards and be interconnected using point-to-point wiring. In order to satisfy physical space limitations, PEs of a larger system would need a different implementation (e.g., a single-board PE) and a more regular interconnection scheme.

Section II describes the PASM design. The prototype implementation and the lessons learned from it are discussed in Section III. In Section IV, specific performance goals are outlined for a 1024-PE PASM system and the hardware required to achieve them is detailed. Special attention is given to physical system layout and interconnection.

II. PASM Overview

A block diagram showing the basic components of PASM [17] is given in Figure 2. The *System Control Unit (SCU)* is a conventional computer and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit* contains N PEs and an interconnection network. The *Memory Management System (MMS)* controls the loading and unloading of the Parallel Computation Unit memory modules from the multiple secondary storage devices of the *Memory Storage System*. The *Micro Controllers (MCs)* are a set of Q microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each MC controls N/Q PEs. An MC together with its N/Q PEs is called an *MC-group*. *Control Storage* contains the programs for the MCs.

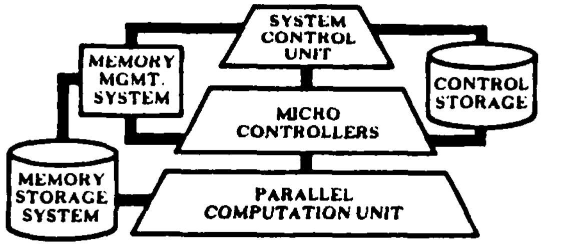


Figure 2. PASM block diagram.

PASM is being designed to have N=1024 PEs and Q=32 MCs. The PASM prototype, illustrated in Figure 3, is being constructed with N=16 PEs and Q=4 MCs.

Virtual Machines

The MCs are the multiple control units needed in order to have a partitionable SIMD/MIMD system. There are $Q=2^q$ MCs, physically addressed (numbered) from 0 to Q−1. The physical addresses of the N/Q processors which are connected to an MC all have as their low-order q bits the physical address of that MC. A virtual SIMD machine of size MN/Q, where $M=2^m$ and $0 \leq m \leq q$, is obtained by having M MCs use the same instructions and synchronizing the MCs. The physical addresses of these MCs must have the same low-order q−m bits so that all of the PEs in the partition have the same low-order q−m physical address bits. Similarly, a virtual MIMD machine of size MN/Q is

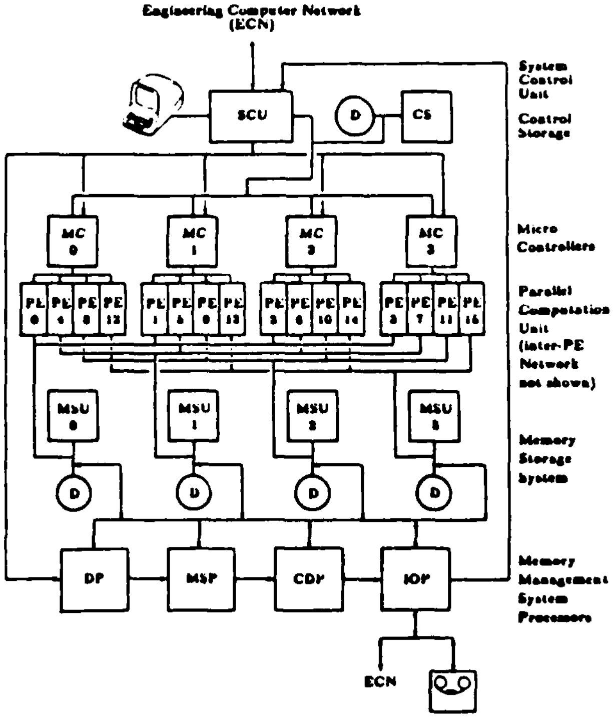


Figure 3. PASM prototype overview.

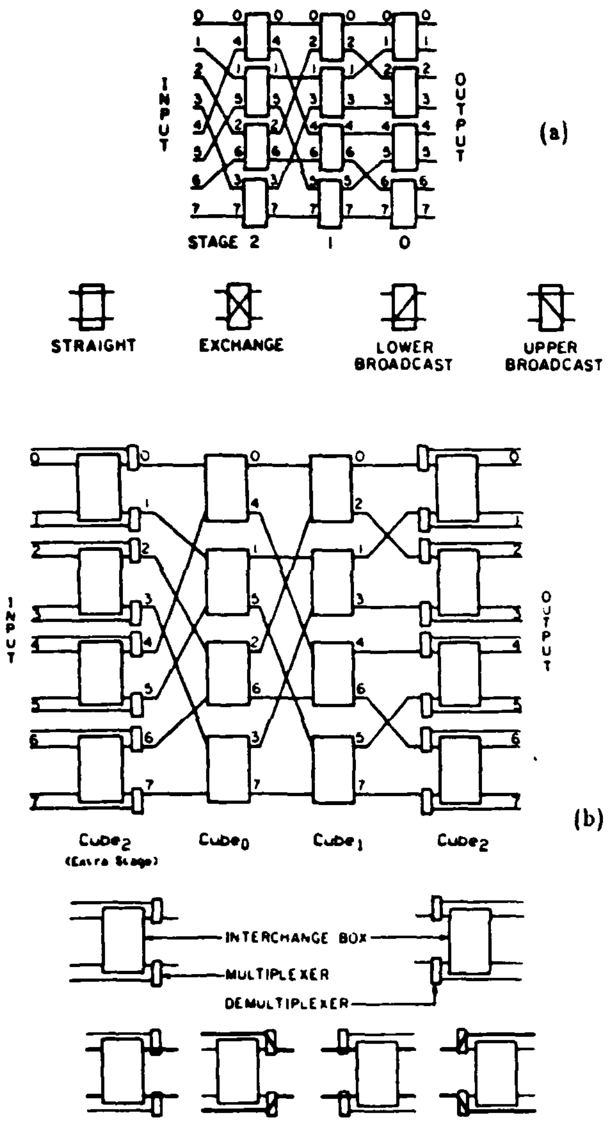


Figure 4. (a) The Generalized Cube with N=8 and the four states of an interchange box. (b) Extra Stage Cube network with N=8.

obtained by combining the efforts of the PEs associated with M MCs which have the same low-order q−m physical address bits. In MIMD mode, the MCs are used to help coordinate the activities of their PEs. Q is the maximum number of partitions allowable, and N/Q is the size of the smallest partition.

Interconnection Network

The interconnection network is an N-input N-output switch that is controlled by the PEs in a distributed fashion. PE i, $0 \leq i < N$, is connected to input port i and output port i of the unidirectional network. The Generalized Cube network topology [20] being used in PASM has $n = \log_2 N$ stages of N/2 interchange boxes. Each interchange box is a two-input, two-output device which can be set individually to one of the four legitimate states shown in Figure 4a.

The connections in this network are based on the cube interconnection functions. Let $P = p_{n-1} \cdots p_1 p_0$ be the binary representation of an arbitrary I/O line label on an interchange box (Figure 4a). Then the n cube interconnection functions can be defined as:

$$\text{cube}_i(p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_1 p_0$$

where $0 \leq i < n$, $0 \leq P < N$, and $\bar{p}_i$ denotes the complement of p_i. This means that the cube_i interconnection function connects P to $\text{cube}_i(P)$ *where* $\textit{cube}_i(P)$ *is the I/O line whose* label differs from P in just the i-th bit position. Stage i of the Cube topology contains the cube_i interconnection function, i.e., it pairs I/O lines that differ only in the i-th bit position.

A network interchange box is controlled by a routing tag which is the destination port address D [21]. Let $d_{n-1} \cdots d_1 d_0$ be the binary representation of D. An interchange box at stage i need only examine d_i. If $d_i = 0$, a connection is made from the interchange box input to the upper output link; otherwise, a connection is made to the lower output link. Thus, if the tag bits associated with a given interchange box are 0 on the upper input link and 1 on the lower input link, the box is set to the straight state. Similarly, if the tag bits are 1 on the upper input link and 0 on the lower input link, the box is set to the exchange state. Other combinations, e.g., 0 tag bit on both the upper and lower input, create "conflicts" in the network since no configuration of the box can make the desired connection.

Tags that can be used for broadcasting data are an extension of this scheme. An n-bit broadcast tag (B) indicates in what stages the boxes are to be placed in a broadcasting mode. For example, if bit b_i is a 1, a broadcast is performed; otherwise, the straight or exchange function specified by the normal tag is used.

Secondary Storage

Each of the secondary storage units in the system consists of a high-capacity Winchester-technology disk drive, disk controller, and a microprocessor to manage the file directory system on the disk. *Control Storage* contains the programs *for the MCs. The loading of programs from Con*trol Storage into the MC memory units is controlled by the SCU. Each MC has two dual-ported memory units so that memory loading and computations can be overlapped.

The *Memory Storage System* provides secondary storage space for the PE data files in SIMD mode and for the PE data and program files in MIMD mode. It consists of N/Q independent *Memory Storage Units (MSUs)*, numbered from 0 to (N/Q)−1. Each MSU is connected to Q PE memory modules. For $0 \leq i < N/Q$, MSU i is connected to those PE memory modules whose physical addresses have the value i in their n−q high-order bits. Recall that, for $0 \leq k < Q$, MC k is connected to those PEs whose physical addresses have the value k in their q low-order bits. This is shown *for* N=16 and Q=4 in Figure 3.

A virtual machine of MN/Q PEs requires only M parallel block loads if the data for the PE memory module whose high-order n−q logical address bits equal i is loaded into MSU i. This is true no matter which group of M MCs (which agree in their low-order q−m physical address bits) is chosen. Each PE has two dual-ported memory units with arbitrated access to allow data to be moved between one memory unit and secondary storage while the processor operates on data in the other memory unit.

Memory Management System

The *Memory Management System* controls the transferring of files between the Memory Storage System and the PEs. It is composed of a separate set of microprocessors dedicated to performing tasks in a distributed fashion. This distributed processing approach is chosen in order to provide *the Memory Management System* with a large amount of processing power at low cost. The Memory

Management System consists a Directory Lookup Processor (DP), a Memory Scheduling Processor (MSP), a Command Distribution Processor (CDP), and an Input/Output and Reformatting Processor (IOP). The division of tasks chosen is based on the main functions which the Memory Management System must perform, including: (1) generating tasks based on PE load/unload requests from the SCU; (2) scheduling Memory Storage System data transfers; (3) controlling input/output operations involving peripheral devices and the Memory Storage System; (4) maintaining the Memory Management System file directory information; and (5) controlling the Memory Storage System bus. User programs and data can be received from or sent to peripherals such as additional mass storage or image input/output devices.

System Control Unit

The SCU is responsible for the overall coordination of the activities of the other components of PASM. The types of tasks the SCU will perform include program development, job scheduling, and coordination of the loading of the PE memory modules from the Memory Storage System with the loading of the MC memory modules from Control Storage.

III. PASM Prototype Design

Physical description

A prototype of the PASM system is currently being constructed (see Figure 3) [10]. All processors in the system are based on the Motorola MC68010 16-bit microprocessor. The use of off-the-shelf components eliminates the need for custom chip designs and thereby reduces development time and construction costs. By using a modular design concept, only eight different types of physical boards are needed:

CPU board -- Motorola MC68010 microprocessor [22] with a VME bus interface [23], on-board EPROM and static RAM, serial I/O port;

Memory board set -- 256KB-2MB dynamic RAM, dual-ported with parity, logical to physical address mapping, and access protection (two physical boards, always used together);

Standard I/O board -- serial and parallel I/O ports for inter-CPU communication, Direct Memory Access (DMA) controller, and performance analysis hardware (all boards of this type); MC/PE communication hardware (boards of this type used in PEs and MCs); instruction broadcast interface and floating point processor (boards of this type used in the PEs),

PE-Network Interface board -- parallel I/O ports and specialized hardware for interconnection network communication;

Network Interchange Box board -- 2-input 2-output switching element;

MC board -- specialized functions for the control of SIMD programs and a 64KB dual-ported static memory for storage of SIMD PE instructions;

Disk Controller board -- controller for Winchester-technology and floppy disks;

Disk Access Switch board -- connects an MSU to multiple PE memory units or Control Storage to multiple MC memory units.

The SCU, MCs, PEs, Memory Management System processors, Control Storage and MSU processors, and the PE interconnection network can all be implemented by using these boards in different configurations. The CPU and Disk Controller boards are commercial products, the other boards are developed in-house. The physical boards are in the standard double Eurocard format; they are connected through the VME bus, which is particularly suited for the Motorola 68000 microprocessor family. A standard bus has the advantage of readily-available backplanes, card cages, and accessories.

The SCU for the prototype is a dedicated microprocessor responsible for the overall orchestration of the activities of the system. It consists of a CPU board, a Memory board set, and several Standard I/O boards. It shares its mass storage device, Control Storage, with the MCs. In order to allow a larger group of users to access PASM, the SCU will serve as a link between PASM and the Engineering Computer Network (ECN). ECN is a local network of about twenty DEC VAX and PDP-11 computers at Purdue University. The user's terminal will be physically connected to an ECN host computer. The host will provide the environment for the development, compilation, and debugging of SIMD and MIMD programs to prevent the SCU microprocessor from being burdened. Commands (jobs) initiated by users are sent by the host to the SCU, which schedules the jobs to be run on the parallel machine.

An MC is composed of a CPU board, a Memory board set, a Standard I/O board, and an MC board containing specialized logic and high-speed memory for time-critical MC functions. One of the specialized components of the MC is the *Fetch Unit*. It is a finite-state machine controlled by the MC CPU that fetches PE instructions from the *Fetch Unit Memory* and broadcasts them to the PEs (when operating in SIMD mode). Together, the double-buffered MC CPU memory (consisting of the Memory board set connected via the VME bus to the CPU board) and the double-buffered Fetch Unit Memory (on the MC board) appear as one logical double-buffered memory to the Control Storage disk. The instructions of an SIMD program are divided among these two physical memories: program flow and control instructions executed by the MC CPU as well as instructions to control the Fetch Unit are contained in the one of the double-buffered MC memory units; PE instructions are contained in one of the Fetch Unit Memory units. Whenever a block of one or more PE instructions is to be executed, the MC CPU sends the block's starting address and length (number of words in the block) to the Fetch Unit. The Fetch Unit carries out the fetching/broadcasting duties in parallel with subsequent MC CPU operations, thus improving throughput. PE instructions that are fetched are queued in a FIFO buffer before being broadcast to the PEs. This allows the actions of the MC to be overlapped with those of the PEs.

Another specialized component of the MC is the *Masking Operations Unit*. It is controlled by the MC CPU and is used to selectively enable and disable sets of PEs. It is also used to examine the global status of PEs; for example, "if any" or "if all" PEs have encountered a certain condition. The output of each MC's masking operations unit is an N/Q-bit *mask* that specifies which PEs are to enabled/disabled. When PE instructions are enqueued in the FIFO buffer, the current mask is also enqueued. When the PEs dequeue the instruction, the corresponding mask is used to enable/disable them.

A PE consists of a CPU board, a Memory board set, a Standard I/O board, and a PE-Network Interface board. The PE-Network Interface board contains parallel ports that allow the PE CPU to communicate via the interconnection network using conventional load and store instructions. A PE specifies where its data is to be routed by computing the address(es) of the destination processor(s) (PEs are addressed 0 to N−1). The address and broadcast tag is written to a parallel I/O port which instructs the network to set switches to make a connection with the destination address(es). Data transmissions will occur through two MC68230 parallel I/O ports called Data Transfer Registers (DTRs) [17]. The two ports are configured as 16-bit unidirectional channels. One direction connects the PE to the network input (DTRin), and the other to the network output (DTRout). The data to be transmitted is written to the DTRin port. This signals to the network that the transfer should be made. Subsequent transfers route items to the same destination until the network setting is changed. In SIMD mode, all PEs do these operations at the same time. In MIMD mode, PEs use the network asynchronously. The data transfers can be done either by using the PE CPUs to transfer each word individually or by setting up the DMA controllers on each end of the network to perform the transfer. Access to a global shared memory is emulated by providing each PE with a memory server process that is activated by memory request messages. A request for a remote data object causes a memory request message to be sent to the remote PE. The memory server process on the remote PE obtains the data and returns it to the requesting PE.

The MC/PE communication controller is found on the standard I/O board. This controller allows data exchange between an MC and its PEs via a shared bus. This data link is necessary to allow the MC to coordinate PE activities in MIMD mode and to allow the PE to report unusual status conditions in SIMD mode.

The PASM prototype's interconnection network is a circuit-switched implementation of the Extra Stage Cube network [24, 25] which is a fault-tolerant version of the Multistage Cube network. This network is single-fault tolerant and has been shown to be very robust under multiple faults [26]. This is because an extra stage at the input end of the network implementing the $cube_{n-1}$ function and bypass switches around the input and the output stages can be used to provide two disjoint paths through the network (Figure 4b). Therefore, when a fault occurs, the alternate path may be used.

In PASM, partitioning is based on PE addresses within a virtual machine agreeing in some number of low-order bit positions. The ordering of the stages as shown in Figure 4b allows this partitioning to occur at the input end of the network which has been shown to reduce the path set-up time [27]. Circuit switching was chosen for its ease of implementation as well as its particular suitability (when compared to packet switching) for the anticipated large SIMD data transfers under DMA control. Using a circuit switched network, prior to any message transmission between a network source-destination pair, a physical path through the network must be made connecting the pair. This path is established through the use of a request-grant protocol. This connects the source-destination pair for the duration of the message transmission Individual words within the message are transferred from the source PE to the destination PE using a handshaking protocol between the parallel ports that interface the PEs to the network.

The PASM prototype network is being constructed from readily available SSI/MSI integrated circuits. It consists of $\log_2 N+1$ (=5) stages of interchange boxes, with eight boxes per stage. A path through the network can be established in approximately 1000 ns (assuming no delays due to network conflicts). The data itself can be transmitted from a network input port to a network output port at a rate of one 16-bit word every 400 ns. With the PEs themselves operating on a 10 Mhz system clock, these transfer times are fast enough so that the network will not act as a bottleneck in the computation process under execution.

Each of the MSUs and Control Storage consists of a CPU board, a Memory board set, a Disk Access Switch Board, a Disk Controller board, and a Winchester-technology disk unit. Control Storage also has a floppy disk unit used when booting the system. The CPU serves as an intelligent disk server handling requests to load/unload files into specified PEs or MCs. The Disk Access Switch board connects the MSU to the Q PEs it serves and provides Q DMA channels to control the I/O transfers. Data may be buffered and reformatted by the CPU as it is accessed from the disk to more effectively utilize the bandwidth of the Q channels between the MSU and the PEs.

Lessons from the prototype

The number of physical boards required to build the PASM prototype is summarized in Table 1. For comparison, the number of boards required to build a PASM system of 1024 PEs and 32 MCs using the same technology is given. Clearly, the implementation scalability of the PEs and interconnection network is poor; the MC and MSU implementations are less critical, but improvements need to be sought.

Table 1. Physical board requirements using prototype implementation technology.

COMPONENT	GENERAL EXPRESSION	PROTOTYPE IMPLEMENTATION	FULL PASM IMPLEMENTATION
PE	$5N$	80	5120
NETWORK	$\frac{N}{2}(\log_2 N+1)$	40	5632
MC	$6Q$	24	192
MSU	$6\frac{N}{Q}$	24	192

PEs. There are clearly too many boards per PE in the prototype for implementation scalability. Some of the problem areas in the prototype PE implementation will now be discussed. The high board count is due in part to the double Eurocard VME bus standard which calls for a rather small board area. Another problem is the path width of the interconnection network: 16 data bits, 2 parity bits, and several control lines are required for both the paths from the PE to the input of network stage n and from the output of stage 0 back to the PE. The situation is further exacerbated by an additional fully redundant path between the input and output of the network and the PE DTRs (Figure 4b). This results in a complete board being used for this PE-network interface.

Separate data paths are used between the MC and the PEs for broadcasting instructions and for the asynchronous MC/PE communication. Further, the MC/PE communication uses parallel data transfers. A great deal of board area and connector space is dedicated to this communication.

The double-buffered dynamic memory board set is complicated by the need to arbitrate the asynchronous requests arriving from the CPU and disk unit (MSU or Control Storage) and by the need to refresh the dynamic memory. In addition, there is board space dedicated to memory management and protection hardware. The 16-bit data and 23-bit address paths between the disk unit and PE are also taking up too much board area. The memory lacks error detection and correction (ECC) logic to allow recovery from single memory errors.

The power consumption per PE is high due the large number of data and address line drivers. The drivers also induce propagation delays that slow data transfers over the

VME bus. Reducing the PE to one board would eliminate the bus drivers, speed data transfers, and reduce the overall chip count.

Network. The other component for which the prototype implementation is not scalable is the interconnection network. Each Network Interchange Box board implements a 2-by-2 crossbar that is 16 data bits, 2 parity bits, and several control bits in width. A substantial portion of the board is dedicated to lines rather than chips. Not enough experience has been gained to determine if the redundant CPU/network connections significantly enhance the overall system reliability.

MCs and MSUs. Board count reductions in the MCs and MSUs are less critically needed, but will result automatically if appropriate steps are taken in the PEs to reduce data path width and to place CPU, memory, and communications on the same board. A high-speed I/O channel allowing direct access to the PE memories is highly desirable for real-time processing

IV. Implementation of a 1024-PE PASM System

Because image processing is one of the primary forces driving the PASM "design philosophy," some performance goals should be set before arbitrarily deciding that 1024 PEs are "enough." One goal should be the processing of 1K-by-1K monochrome images (one byte per pixel) at TV rates (30 images/sec). The processing should be a significant enhancement algorithm such as contrast enhancement or edge sharpening. The performance of such algorithms is dominated by the time spent in fetching pixel values from memory. Thus by counting the number of times each pixel is fetched, a rough estimate of the number of instructions the machine must execute per second can be derived. Suppose that a good edge sharpening algorithm uses 5-by-5 pixel "windows" which must be compared at each "match position" in the image to enhance the edge. Each pixel would then be fetched up to 25 times, once for each "match position." If 20 machine cycles are required to fetch each pixel (appropriate for 68000-family processors, including requisite index calculations), 500 machine cycles are expended on processing each pixel. Since 1K-by-1K times 30 pixels must be processed per second (30M pixels/sec), the machine must provide 15 Giga-cycles per second. Current microprocessors in the 68000 family have 12.5 to 16.7 MHz clocks, making 1024 processors sufficient to achieve this processing rate. If more complex algorithms are to be executed, the number of PEs or speed of the CPUs can be increased.

Another goal determines the desired memory size for each PE. Satellite images approaching 5K-by-5K pixels are now being considered by those involved in digital photogrammetry. Often these images have data from several sensors, resulting in storage requirements of several bytes per pixel. Assuming four bytes/pixel, 100M bytes of memory are required. If each of the 1024 PE memory units had a capacity of 128K bytes and the image were distributed evenly among the PEs, five bytes per pixel could be stored in each PE. This could be the entire image as well as a "working image" of one byte per pixel. (The memories of each of the 16 prototype PEs are larger so that reasonably-sized images can be processed.)

In scaling up the number of PEs to 1024, the number of MCs was set at 32. This decision is made for practical reasons as 32 represents reasonable limit on the number of PEs that could be placed close enough together to be interconnected by a bus.

Another constraint is that in scaling up to 32 MCs, the number of MSUs increased to 32 and the number of PEs serviced by an MSU became 32. Assuming a fixed subimage size, the amount of data handled by each MSU has increased eightfold since each MSU now serves 32 PEs instead of the 4 PEs each served in the prototype. Of more concern is the eightfold increase in overhead for servicing the load/unload requests (e.g., searching the disk volume directory structure). This suggests that 32 MSUs might not be enough when the machine is operating in MIMD mode (in SIMD mode, the disk volume can be formatted such that all of the data is in one file, minimizing the increase in the overhead). More experience with the prototype will be necessary before determining if an MSU servicing four PEs is appropriate or whether it represents excess capability.

Using the previous discussion of the limitations of the prototype and the design constraints given above, the design of a 1024-PE PASM system will now be outlined. LSI (1000 to 100,000 transistors) and VLSI (more than 100,000 transistors) approaches will be used to reduce the chip count, allowing more functions to be packed on each physical board. Use of LSI and VLSI chips will also reduce power consumption and increase reliability by internalizing interconnections. We advocate the the use of commercial LSI and VLSI building blocks wherever possible because these are the result of hundreds or thousands of man-years of effort which cannot be duplicated except at extreme cost. The functions peculiar to parallel processing that are required for this design are to be implemented by components external to the commercial building blocks. We do not propose to reduce chip counts by integrating these specialized functions into existing LSI and VLSI building blocks, even though it is feasible from a technological standpoint, because of high cost and the unwillingness of semiconductor manufacturers to share their high-performance designs. Rather, custom LSI approaches for the specialized functions are suggested where the need for space savings is greatest and where no commercial part is suitable.

Figure 5 shows the proposed PE organization for such a system. It consists of a 16.67 MHz Motorola MC68020 CPU, a Motorola MC68881 floating point co-processor, two dual-ported 128KB (or larger) static memories with ECC logic, EPROM, a serial interconnection network interface, and a 32-bit parallel (plus control bits) interface to the MC. The CPU has an on-board instruction cache that enhances the PE's performance in MIMD mode. In SIMD mode, the PE gets its instructions from the MC through the MC interface; therefore the instruction cache is not used in SIMD mode. The MC interface allows bi-directional communication: in SIMD mode, instructions are broadcast to the PEs; in both SIMD and MIMD mode, PEs can request one-to-one communication with their MC to report their status or to alert an error condition.

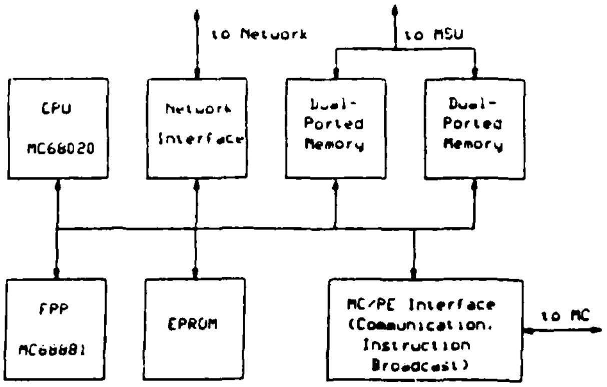


Figure 5. PE organization.

The MC68020 CPU and MC68881 floating point coprocessor represent a use of commercial VLSI products. Even though there are some functions peculiar to parallelism that could be implemented in the CPU, a custom design would be unlikely to achieve the performance of these products. However, the serial interconnection network interface on the PE boards is an area where a custom LSI chip would be useful. It would construct data packets (this does *not* imply a packet-switching network), add parity or cyclic redundancy code bits to ensure correctness, and interact with the interconnection network. It could also handle the low level aspects of a network protocol [28] such as character escaping, generation of acknowledgements and retransmit signals, and monitoring of "timeout" conditions.

Frequently in image processing algorithms, a PE must transfer its *subimage* (image subdivision per PE) edge pixels to neighboring PEs. The use of DMA for the "block transfer" of this data is desirable when the size of the blocks is sufficiently large to compensate for the setup overhead of the DMA controller. The MC68020 CPU can perform the actions of the DMA controller fairly efficiently because the CPU instructions in the "tight loop" implementing the transfer can be cached. Also, the CPU can perform a much wider range of transfers than can a DMA controller. For example, if subimage data is stored in memory in row-major order, most existing DMA controllers cannot transfer a subimage column as a block. Therefore, the use of DMA seems limited to a small number of applications unless a "super" DMA controller that could perform the specialized types of transfers motivated by image processing applications could be designed. Whereas the complexity of current DMA controllers is comparable to that of VLSI CPUs, we do not propose to include a "super" DMA controller in the design for cost reasons. Furthermore, we are aware of no existing 32-bit DMA controllers.

Savings in board area as compared to the prototype implementation are due to the use of a smaller (physical and logical size) memory, static RAM chips (needing no refreshing hardware), simplified memory access control, multiplexed functions of the MC interface, a serial network interface, and reduced numbers of line drivers (no VME bus used). Use of grid-array and surface-mount technologies also markedly increase the density of the layout. Figure 6 shows the proposed PE board layout. The MC-PE interconnection bus connector mates with a horizontal backplane bus that interconnects the MC-group. As shown in Figure 7, an MC-group is arranged in a horizontal slice with 16 PEs on each side of the 3-board implementation of the MC. This arrangement simplifies the driving of horizontal backplane busses and reduces propagation delays. Eight such MC-groups are arranged vertically in a cabinet to form the quadrant shown in Figure 7. Four such quadrants comprise the set of MCs and PEs.

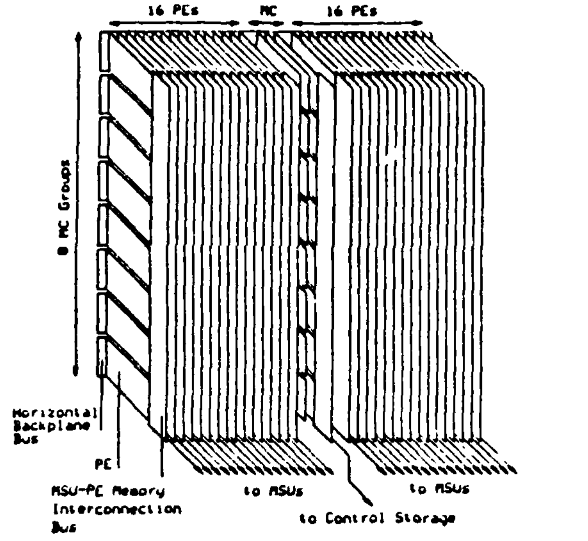


Figure 7. Physical organization of a PASM quadrant containing 256 PEs arranged in 8 MC groups.

The logical PE i within each MC-group is serviced by MSU i. Figure 7 shows that the MSUs are connected to the PEs through a vertical bus. The MSU sees the memories of the PEs it services as as a contiguous block; upper address bits select which PE's memory drivers are enabled for a data transfer.

The MSU organization is shown in Figure 8. Its organization is similar to that used in the prototype except that an External I/O Interface allowing connection to real-time I/O devices is added. Such an input channel to each MSU gives direct access to the PE memories by bypassing the secondary storage media. Use of the Buffered Interfaces (Figure 8) makes 4-way memory interleaving possible when accepting real-time data, enhancing throughput. With 32 MSUs each having the 4-way interleaved channels, there are 128 channels on which 32-bit data can be transferred. Assuming a 200 ns access time of the PE memories, an

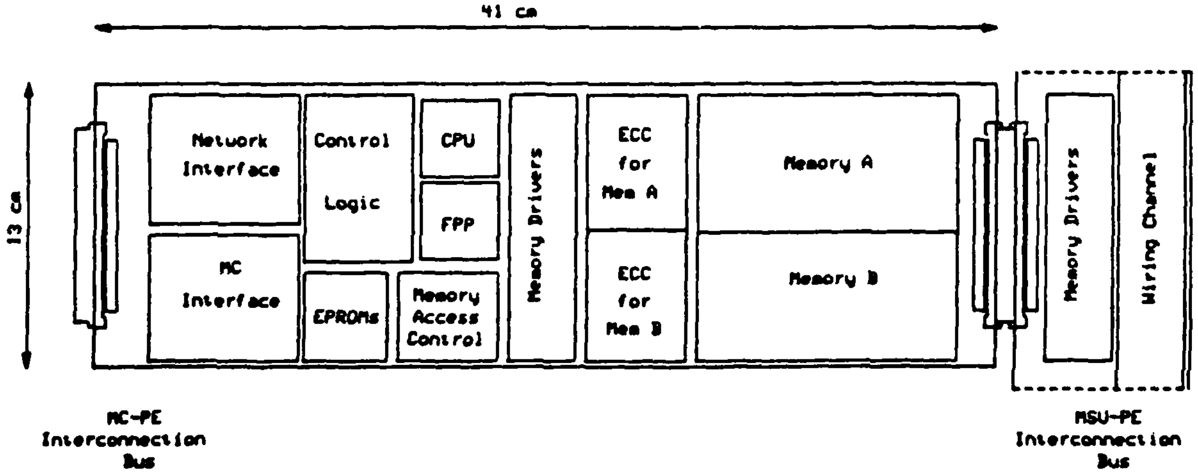


Figure 6. PE board layout for a 1024-PE PASM system.

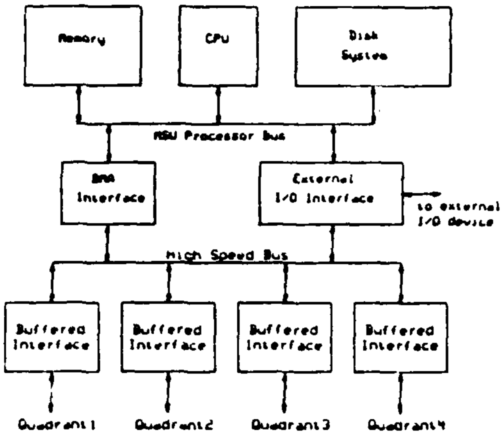


Figure 8. MSU structure for a 1024-PE PASM system.

overall input speed to PASM in excess of 500 million 32-bit words per second can be achieved.

The most radical departure from the original prototype design occurs in the interconnection network. First, the 16-bit data path width in the prototype design which limited each physical VME-standard board to the implementation of a 2-by-2 interchange box could not be justified in a 1024-PE PASM system: over 5000 boards would have to be interconnected. Reducing the width to eight or four bits does not make enough of an impact on the number of boards required because at most a 4-by-4 interchange box could be fit on a single standard-sized board resulting in more than 1000 boards. The only practical solution seems to be a bit-serial network where the functions of a 2-by-2 or 4-by-4 interchange box would be integrated on one custom LSI chip.

When bit-serial networks are being considered, the choice between circuit- and packet-switching is not clear-cut. Conflicts are more common and the delays due to establishing a network path and propagating acknowledgement handshake signals are increased for large circuit-switched networks. On the other hand, once a circuit has been established, the throughput of the network is higher than that for packet switching. Because the most appropriate communication mode is application dependent, designing the network to operate in both modes is desirable. A similar "hybrid" circuit/packet switched network approach was suggested in [29].

To simplify the design of the network protocol and hardware, two bit-serial lines are used for each interconnection link between the output of one interchange box and the input of another. This is to avoid the logic that would be necessary to multiplex the lines between the functions of setting the network (interpreting the data as routing tags) and using the network to transfer data. The internal architecture of an LSI chip implementing an interchange box of the "hybrid" Extra Stage Cube network will now be described. Work is currently underway at Purdue University on the design of such a chip.

A 2-by-2 interchange box is organized as shown in Figure 9. The protocol governing packet-switched transfers through an interchange box in stage i is stated as follows: if a box in the previous stage (stage i−1) is connected to this box by the upper (lower) data line and has data to transfer to this box, the box in the previous stage asserts a request on the upper (lower) handshake line. If the box in stage i has an empty upper (lower) Input Queue, it responds with a grant signal on the upper (lower) Handshake line. The data packet arrives on the Upper Input (UI) or Lower Input (LI) line and is shifted into an empty Input Queue. Simultaneously, a control packet associated with the data packet is shifted into a corresponding control register. The Interchange Box Control examines the Routing (R) and Broadcast (B) bits (part of the control packet) that control the ith stage and determine to which output(s) the data is to be routed. Handshake line(s) requesting the desired output(s) are asserted. When a granting signal is received from the next stage (stage i+1), the 2-by-2 crossbar is set and the data and control packets are shifted out of the Input Queue and into the next stage. The Interchange Box Control is a finite state machine that arbitrates requests and handles conflicts.

In circuit-switching mode, the PEs first generate control packets which propagate through the network establishing connections. The inter-stage request/grant protocol controlling the movement of the control packets is exactly as described above except that there are no accompanying data packets. When the complete path has been established, there is a virtual circuit formed between the input and output ports of the network and data packet transfers may begin. The Data Input Queues are bypassed in circuit-switching mode.

Two possibilities to implement a 4-by-4 subnetwork chips exist. Either four of the 2-by-2 interchange boxes could be interconnected on a chip, or a chip with eight instead of four Data Input Queues and with a 4-by-4 instead of a 2-by-2 crossbar switch could be designed. Studies [30, 31] indicate that the performance of a design with a

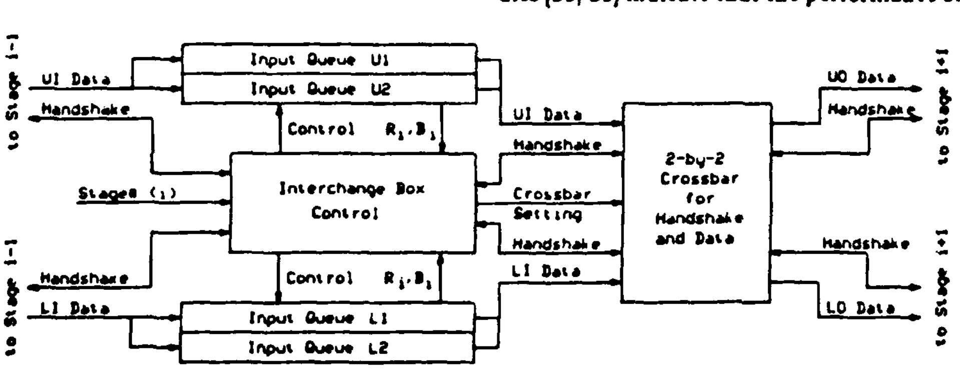


Figure 9. 2-by-2 network interchange box. The input Handshake lines are comprised of a request signal from and a grant signal to stage i−1. The output Handshake lines consist of a request line to and a grant line from stage i+1.

4-by-4 crossbar will have higher performance; the LSI design, however, will be more complex. For both designs, each network input/output link consists of four signals (data, control, request, grant). Other chip connections required are power, clock input, reset, and stage number input. Another input on the 4-by-4 design would be used to bypass one of the two internal stages. This would allow the chip to be used in forming larger subnetwork modules that have a number of inputs/outputs that is not an even power of two. The 2-by-2 design would require a 28-pin DIP if integrated alone; the 4-by-4 subnetwork module would require a 48-pin DIP.

Given that a 4-by-4 subnetwork module can be integrated on a 48-pin chip, a 32-by-32 subnetwork module can be constructed by interconnecting 24 chips on a board (bypassing an internal stage on eight of the chips). Using the interconnection scheme proposed for the Ultracomputer [14], network stages 0 through 4 of the 1024-PE PASM network would be implemented using 32 32-by-32 subnetwork module boards (chips implementing network stage 4 would have their other stage bypassed). An additional 32 subnetwork module boards would implement stages 5 through 9 (chips implementing network stage 9 would have their other stage bypassed). This is shown in Figure 10. In order to provide fault tolerance, both stages implementing the $cube_0$ function must be independent of other cube functions. All the PEs in an MC-group can be connected to the same 32-by-32 subnetwork as shown in Figure 10. Thus, all of the extra stage circuitry (4-by-4 chips and extra stage bypass multiplexers) can be physically situated on the horizontal backplane busses. (The extra stage circuitry is not shown in Figure 7.) The 1024 output links of the extra stage are connected to the 32 32-by-32 subnetwork modules that implement stages 0 through 4 of the network. The 2018 output links of the 32 32-by-32 subnetwork modules that implement stages 5 through 9 of the network are routed back to the PEs. Twice as many links between the output of the network and the PEs are required for redundancy. Still, since each cable "link" consists of only five wires (data, control, request, grant, and ground), this cable bulk will be manageable. Redundant independent links were not deemed to be necessary between the PEs and the input to the extra stage because the horizontal backplane bus connections are much more reliable than the discrete connecting cables.

Due to the serial approach, speed of the network will be limited. For packet-switching, if a 20 MHz clock speed and a queue length of 64 bits for the network chips are assumed, a message can be strobed into a chip in approximately 6.4 μs. To reach the destination, a message has to pass through $\log_2 N + 1 (=11)$ stages; travel time through the network will therefore be approximately 71 μs. Since messages are queued in every stage, a pipeline effect will result in steady state; assuming no conflicts, packets of a long message will arrive in 6.4μs intervals. Circuit-switched performance is better since the messages encounter only the propagation delays of the internal 2-by-2 crossbar switches which would be no more than 20ns per stage (approximately 500ns for end-to-end data transmission and acknowledgement return). However, the circuit must be established before it is used. This requires approximately 35 μs for 32-bit control packets if there are no conflicts. This is about half of the time required for an end-to-end 64-bit data packet movement.

The 3-board set implementing an MC in a 1024-PE PASM system consists of a CPU/memory board; a board for the Fetch Unit, its memory, and Masking Operations Unit; and an I/O board. The CPU/memory board would be similar to the PE board (Figure 6) except that the area for the floating point co-processor, Network Interface, and MC-PE Interface would be used for additional memory and a bus interface to the other MC boards. The two other MC boards are scaled-up versions of those used in the prototype.

The floor plan of the proposed system is illustrated in Figure 11. The network cabinets contain stages 0 through 9 of the Extra Stage Cube network. The extra stage itself is physically situated with the MC-groups. The SCU may be a stand-alone microprocessor system attached to a host as is used in the prototype or may be a super-mini computer such as a DEC VAX.

It is estimated that the equipment cost for a 1024-PE PASM system would be $2.7 million; $1.5 million for the MCs and PEs, $0.5 million for the interconnection network (including fabrication cost of the custom LSI chips), $0.5 million for the MSUs, and $0.2 million for the SCU and Memory Management System processors. Development time including custom LSI chip design is estimated to be approximately ten man-years.

V. Summary

A design of a PASM system consisting of a 1024-microprocessor computational engine, multiple secondary storage units, and distributed memory management and control processors has been proposed. Use of off-the-shelf CPUs, memories, and other components in the design of the 30-processor PASM prototype was found to provide the necessary functionality and performance of that system. Their proposed use in the full PASM system maximizes the performance/cost ratio because the development cost of

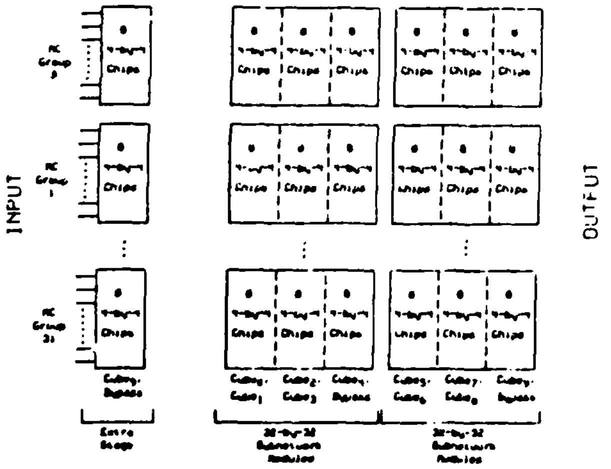


Figure 10. Extra Stage Cube network layout for the 1024-PE PASM system.

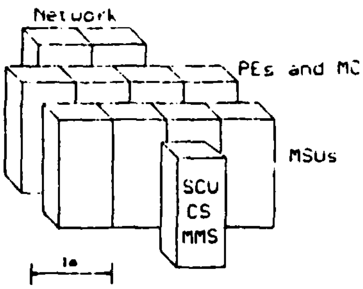


Figure 11. 1024-PE PASM system floor plan. CS is Control Storage; MMS represents the Memory Management System processors.

these commercial LSI and VLSI components has been amortized over many more units than would be produced by a custom PASM design. Only where space savings are critical and/or the specialized nature of the component makes the functions unavailable commercially have custom LSI approaches been incorporated in the design. Each proposed custom LSI chip design has been examined and determined to be within the capabilities of existing in-house design talent and CAD tools.

The PASM architecture, like MPP, achieves its high performance from a sheer multiplicity of computing resources. It does so without the reliance on ultra-sophisticated technologies and hand-crafted "tuned" interconnections found in the vector supercomputers such as the Cray-1 and Cyber-205. Being reconfigurable and capable of both SIMD and MIMD operation, it can be used effectively for a larger class of applications than can an SIMD-only machine such as MPP. For vector processing, with 1024 PEs, it has more potential computational power than do existing commercial vector/pipeline computers. In addition, it has more versatility than a conventional vector/pipeline computer because the "vector length" (number of PEs for PASM) can be varied. The capabilities of PASM as a multiprocessor are also significant due to its use of sophisticated processing elements. PASM also has a significant amount of fault tolerance inherent in its architecture: one MC group with a faulty PE can be left unused, faulty PEs in a group may be ignored by not scheduling MIMD processes onto them, redundant connections are present in the network, and so on. With this design, it appears that an SIMD/MIMD machine with performance exceeding 1000 MOPS can be readily constructed using current device and interconnection technologies.

Acknowledgement The authors thank Nathaniel J. Davis IV for his valuable comments.

References

[1] W. R. Wittmayer, "Array processor provides high throughput rates," *Computer Design*, Mar. 1978, pp. 93-100.

[2] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM*, Jan. 1978, pp. 63-72.

[3] M. J. Kascic, *Vector Processing on the Cyber 200*, Technical Manual, Control Data Corporation, 1979, 38 pp.

[4] K. E. Batcher, "STARAN series E," *1977 Int'l. Conf. Parallel Processing*, Aug. 1977, pp. 140-143.

[5] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comp.*, Vol. C-29, Sept. 1980, pp. 836-844.

[6] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, Vol. 54, Dec. 1966, pp. 1901-1909.

[7] H. F. Jordan, "Performance measurement of HEP - a pipelined MIMD computer," *10th Symp. Comp. Arch.*, June 1983, pp. 207-212.

[8] L. C. Widdoes, Jr., "The S-1 project: developing high-performance digital computers," *IEEE Comp. Soc. Spring Compcon 80*, 1980, pp. 282-291.

[9] W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. IEEE*, Vol. 60, Apr. 1972, pp. 369-388.

[10] T. J. Fountain, "CLIP4: progress report," in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi, eds., Academic Press, London, England, 1981, pp. 281-291.

[11] K. E. Batcher, "Bit serial parallel processing systems," *IEEE Trans. Comp.*, Vol. C-31, May 1982, pp. 377-384.

[12] W. Wulf and C. Bell, "C.mmp—A multi-miniprocessor," *AFIPS 1972 Fall Joint Computer Conference*, Dec. 1972, pp. 765-777.

[13] R. J. Swan, S. Fuller, and D. P. Siewiorek, "Cm*: a modular multimicroprocessor," *AFIPS 1977 Nat'l. Comp. Conf.*, June 1977, pp. 637-644.

[14] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer – designing an MIMD shared-memory parallel computer," *IEEE Trans. Comp.*, Vol. C-32, Feb. 1983, pp. 175-189.

[15] G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Symp. Comp. Arch.*, Mar. 1977, pp. 147-152.

[16] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 Nat'l. Comp. Conf.*, June 1980, pp. 631-641.

[17] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comp.*, Vol. C-30, Dec. 1981, pp. 934-947.

[18] H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: a reconfigurable parallel system for image processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition*, July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter: *Computer Architecture News*, Vol. 12, No. 4, September 1984, pp. 7-19).

[19] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Comp. Soc. Spring Compcon 85*, Feb. 1985, pp. 429-434.

[20] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1985.

[21] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comp.*, Vol. C-24, Dec. 1975, pp. 1145-1155.

[22] Motorola, Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual (fourth edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[23] Mostek Corporation, Motorola Inc., and Signetics/Philips, *VME Bus Specification Manual*, Revision A, Mostek Corporation, Motorola Inc., and Signetics/Philips, 1981.

[24] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Comp.*, Vol. C-31, May 1982, pp. 443-454.

[25] N. J. Davis IV and H. J. Siegel, "The PASM prototype interconnection network," *1985 Nat'l. Comp. Conf.*, July 1985, pp. 183-190.

[26] G. B. Adams III and H. J. Siegel, "Modifications to improve the fault tolerance of the extra stage cube interconnection network," *1984 Int'l. Conf. Parallel Processing*, Aug. 1984, pp. 169-173.

[27] N. J. Davis IV and H. J. Siegel, "The performance analysis of partitioned circuit switched multistage interconnection networks," *12th Symp. Comp. Arch.*, June 1985, pp. 387-394.

[28] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[29] U. V. Premkumar, R. N. Kapur, M. Malek, G. J. Lipovski, and P. Horne, "Design and implementation of the banyan interconnection network in TRAC," *AFIPS 1980 Nat'l. Comp. Conf.*, June 1980, pp. 643-653.

[30] G. B. Adams III and H. J. Siegel, "The use of 4×4 switching elements in the multistage cube network," *First Int'l. Conf. Computers and Applications*, June 1984, pp. 585-592.

[31] M. Malek, S. Cheemalavagu, M. S. Juang, and D. B. Rathi, *Design, Packaging, Performance and Self-diagnosis of a 4x4 Banyan Interconnection Network*, Department of Electrical Engineering, The University of Texas at Austin, 1982.